第 3 章　栈和队列

栈和队列是两种重要的数据结构，有着广泛的实际应用。栈和队列都是特殊的线性结构，是操作受限的线性表。数组也是软件设计中最常用的数据结构，可视为线性表的变化形式。本章将介绍这三种结构的有关概念、特性、运算、存储结构及相应的运算实现。

3.1 栈

栈是按“先进后出”操作方式组织的数据结构，日常生活中随处可见这种操作组织方式，比如我们洗碗，将碗摞成一堆，先洗好的碗最后才能拿出，又如一些枪支的子弹夹，子弹的压入和弹出也是这种操作方式。在计算机中栈是一种应用广泛的技术，许多类型 CPU 的内部就构建了栈，在操作系统、编译器、虚拟机等系统软件中栈都有重要的应用，比如，函数调用、语法检查等。在应用软件设计和实际问题求解中更是经常会用到栈结构，比如，表达式求解、回溯法求解问题、递归算法转换为非递归、树和图搜索的非递归实现等。可见，栈是一种重要的、应用广泛的数据结构。

3.1.1 栈的定义和运算

1. **基本概念**

栈（stack）是限定只能在一端进行插入和删除操作的线性表。进行插入和删除操作的一端称为**栈顶**（top），另一端称为**栈底**（bottom），如图 3-1 所示。与线性表一样，称没有元素的栈为**空栈**。

我们称栈的插入操作叫**入栈**（push），栈的删除操作叫**出栈**（pop），早期的许多教材中分别称为**压栈**和**弹栈**。

由栈的定义可知，栈是操作（运算）受限的线性表。

下面分析一下栈的**特征**：假设初值为空的栈，按 $e_1, e_2, \cdots, e_n$ 次序，将这些元素依次连续入栈，由定义可知，这些元素的出栈次序只能是 $e_n, e_{n-1}, \cdots, e_2, e_1$，也就是说，栈具有**后进先出**(LIFO—Last In, First Out)或**先进后出**(FILO—First In, Last Out)的特性。

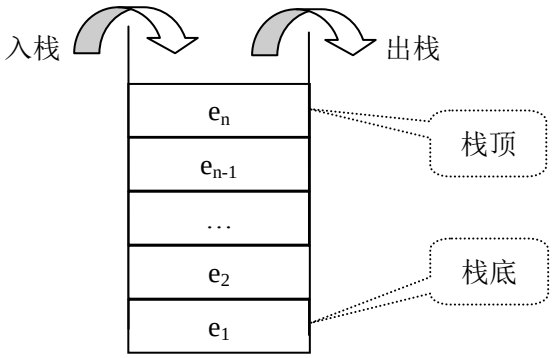


图 3-1 栈结构示意图

2. **栈的运算**

栈的基本运算与线性表的基本运算类似，但又有其特殊之处。常见的基本运算有以下几种：

(1)初始化栈：initialStack(S)

设置栈 S 为空栈。

(2)判断栈是否为空：stackEmpty(S)

若栈 S 为空，返回 TRUE，否则，返回 FALSE。

(3)取栈顶元素值：stackTop(S,x)

若栈 S 不空，将栈 S 的栈顶元素的值送变量 x 中，否则应返回出错信息。

(4)判断栈是否为满：stackFull(S)

栈 S 为满时，返回 TRUE，否则，返回 FALSE。

(5)入栈：pushStack(S,x)

将值为 x 的元素插入到栈 S 中。若插入前的栈已经满了，不能入栈时，应报出错信息。

(6)出栈：popStack(S)

若栈 S 不空，删除栈 S 的栈顶元素，否则应返回出错信息。

上述基本运算中涉及栈满概念的都是针对限定最大元素个数的顺序栈结构。

和线性表一样，这些运算都是基本运算，其具体实现形式可根据具体需求作相应变化。关于栈更为复杂的运算可由这些基本运算复合而成。

下面分别讨论栈的两类存储结构及其在相应结构上的运算实现。

3.1.2 顺序栈

1.**存储结构**

与顺序表一样，以顺序存储方式存储的栈叫做**顺序栈**，可用数组 data[MAXLEN] 的前 n 个元素来存储栈的元素值。在用数组 data[] 存储栈的元素时，数组的哪一端作为栈顶？哪一端作为栈底呢？对这一问题，可能有些读者认为数组的哪一端作为栈顶都可以，但实际上栈顶选择不同，实现的算法的时间性能会有很大差异。

正确的方法是将栈顶放在数组的后面，即 data[n-1]作为栈顶，data[0]为栈底。因为栈的入栈和出栈操作即为线性表的插入和删除操作，当以 data[n-1]为栈顶时，入栈和出栈操作都不需要移动栈中的其它元素。而以 data[0]为栈顶时，入栈和出栈都需要移动栈中几乎全部元素。

此外，需要一个变量用来记录栈顶位置或栈中元素个数，不妨使用变量 top 来指示栈顶。本书中 top 与数组下标一致，即若栈中有 n 个元素，则栈顶 top=n-1，栈顶元素 data[top] 即数组的最后一个元素 data[n-1]。当然也可以用元素的序号来标记栈顶，不同的只是与数组下标差 1。可见由数组 data[]和栈顶指示器 top 就可以唯一确定一个顺序栈，顺序栈的结构如图 3-2 所示。

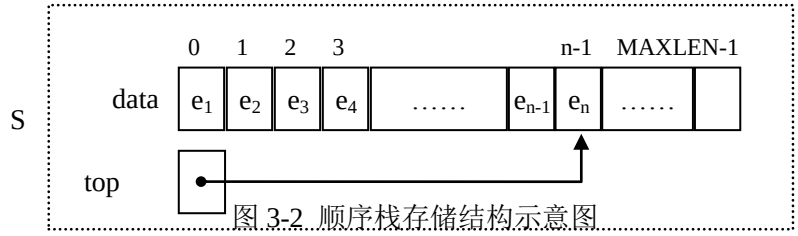


图 3-2 顺序栈存储结构示意图

【顺序栈存储结构描述】

```
typedef struct sStack
{
    elementType data[MAXLEN];  //存放栈元素
    int         top;           //栈顶指示器
} seqStack;
```

【思考问题】分析顺序栈和顺序表存储实现的异同。

2.**顺序栈上运算的实现**

在顺序栈上实现栈的运算与线性表基本相同，因此不再给出分析。

(1)初始化栈

初始化栈就是要将栈设置为空栈，即将其元素个数设置为 0。但顺序栈中没有设置元素个数这一分量，只有栈顶指针 top，而 top 为数组下标，data[0]为栈底元素，因此，top 不能初始化为 0。所以，我们将 top 初始化为-1，即 top=-1，来标记栈空。

【算法描述】

```
void initialStack(seqStack &S)
```

```
{
    S.top=-1;
}
```

(2)判断栈空

即判断 top 的值是否为－1。

【算法描述】

```
bool stackEmpty(seqStack &S)
{
    if(S.top==-1)
        return true;
    else
        return false;
}
```

(3)取栈顶元素

若栈不为空，返回栈顶元素的值，否则返回出错信息。较为通用的方法是将返回的元素以参数的形式给出。

【算法描述】

```
bool stackTop(seqStack &S, elementType &x)
{
    if(stackEmpty(S))
        return false; //空栈，返回 false
    else
    {
        x=S.data[S.top];
        return true;  //取得栈顶，返回 true；取得的值用 x 传递。
    }
}
```

(4)判断栈满

栈顶指针为 top，则栈中元素个数为 top+1，当 top+1==MAXLEN 时，说明栈空间已放满元素。

【算法描述】

```
bool stackFull(seqStack &S)
{
    if(S.top==MAXLEN-1)
        return true;  //栈满，返回 true
    else
        return false; //栈未满，返回 false
}
```

(5)入栈

入栈即在栈顶插入一个元素。插入前需要先判断是否已经栈满，栈满则不能插入，报错。栈不满，栈顶指示器 top 加 1，将元素入栈，。

【算法描述】

```
bool pushStack(seqStack &S, elementType x)
{
    if(stackFull(S))
        return false; //栈满，元素不能入栈，返回 false
    else
    {
        S.top++;          //栈顶后移
        S.data[S.top]=x;  //数据入栈
        return true;      //元素入栈成功，返回 true。
    }
}
```

(6)出栈

出栈即删除栈顶元素。删除栈顶元素前需要先判断是否栈空，若栈空，没有元素可删除，报错。

删除栈顶元素时，可顺便将栈顶元素返回到主调函数，下面的算法在元素出栈的同时，利用函数参数向主调函数返回删除的元素值。当然，也可以不返回删除的元素，因为我们前面已经定义了取栈顶元素函数 stackTop()，需要时可以先调用 stackTop()取栈顶，再出栈。

【算法描述】

```
bool popStack(seqStack &S, elementType &x) //x 返回出栈的元素
{
    if(stackEmpty(S))   //空栈，没元素出栈，返回 false
        return false;
    else
    {
        x=S.data[S.top];   //取栈顶元素至变量 x
        S.top--;           //栈顶减 1，即删除了栈顶元素
        return true;       //出栈成功，返回 true
    }
}
```

【算法分析】这几个算法的时间复杂度均为 O(1)，因此从时间性能方面来说，这种结构是比较理想的。

【思考问题】上面介绍的算法在函数参数传递时，基本上都是用 C++的“引用”实现的，如果换为“指针”，如何实现呢？

顺序栈和顺序表面临着同样的问题，即需要按最大空间需求分配栈空间，而一个实际的软件系统往往需要多个栈，如何合理分配栈空间就成了相对棘手的问题。如果为每个栈都分配很大的空间，可能造成空间利用率不高；分配少了又会经常出现栈溢出。针对此问题，人们想了很多办法，比如两个顺序栈共享空间，还有就是采用链式存储结构，按需分配栈空间。接下来，我们就讨论栈的链式存储实现 – 链栈。

3.1.3 链栈

链栈即采用链式存储结构实现的的栈。链栈可用单链表结构来表示，不失一般性，本书采用不带头结点的单链表结构形式。在这种表示中，同样要注意栈顶位置的选择。我们将链表的首元素结点作为栈顶，尾元素结点作为栈底，用栈顶指针 top 替代链表中的头指针（其实只是叫法不同），链栈结构如图 3-3 所示：

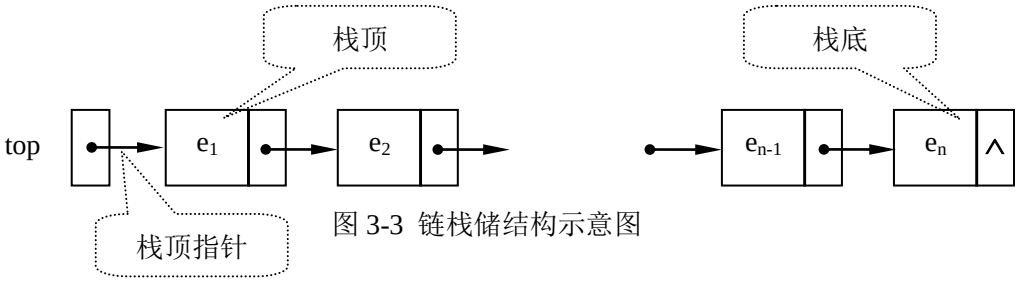


图 3-3 链栈储结构示意图

【链栈存储结构描述】

```
typedef struct lsNode
{
    elementType data;      //链栈结点数据域
    struct lsNode *next;  //链栈结点指针域
} sNode, *linkedStack;
```

在这种结构上实现运算，事实上已变成对链表的运算，入栈和出栈分别变成了在表头插入和删除元素结点的运算，因而是较简单的运算。**链栈没有栈满问题**，下面仅对初始化、入栈和出栈进行描述。

(1)初始化栈

由于不带头结点，初始化仅需将栈顶指针置为空。

【算法描述】

```
void initialStack(sNode *& top)
{
    top=NULL;
}
```

(2)入栈

【算法描述】

```
void pushStack(sNode *& top, elementType x)
{
    sNode* s;
    s=new sNode;     //申请新结点
    s->data=x;       //装入元素值
    s->next=top;     //新结点链接到栈
    top=s;           //更新栈顶指针，使指向新结点
}
```

(3)出栈

下面的算法用参数 x 返回出栈的元素。

【算法描述】

```
bool popStack(sNode *& top, elementType &x)
{
    sNode* u;
    if(top==NULL)
        return false;   //栈空，返回 false
    else
    {
        x=top->data;    //取栈顶元素，由变量 x 返回
        u=top;          //栈顶指针保存到 u
        top=top->next;  //栈顶指针后移一个元素结点
        delete u;       //释放原栈顶结点
        return true;    //出栈成功，返回 true
    }
}
```

以上三个算法中，函数参数中的栈顶指针都用 sNode *& top 进行定义，即栈顶指针的引用，因为这三个函数调用前后，指针变量 top 的值都发生变化，不这样定义，比如仅定义为 sNode* top，子函数中虽修改了 top 的值，主调函数中 top 的值仍保持为调用前的值，不能感知栈的变化。如果不用“引用”，用指针的指针，或函数返回值也可以完成传递任务。这一点在实现时，千万注意。

这里我们只完成了基本运算中的三个算法，**链栈没有栈满的问题**，剩下的判定栈空和取栈顶元素，请读者自行完成。

3.1.4 栈的应用

如前所述，栈是软件设计中最基础的数据结构，有着广泛的应用，其中最具有代表性的应用是用于子程序调用的实现、递归的实现以及表达式的计算。下面简要介绍其应用实例。

1. 栈的基本应用实例

【例 3.1】设计算法完成如下功能：从键盘读入读入 n 个整数，然后按输入次序的相反次序输出各元素的值。例如，依次输入 5 个整数为 1，2，3，4，5，则算法的输出为 5，4，3，2，1。

【解题分析】很显然，输出操作应在读入所有输入的整数之后才能进行，因此，需要设计一个结构存放所读入的数据。选择什么样的结构来存储这些数据呢？许多初学者可能会想到用一个数组，这是不太合适的，因为数组的元素个数是在程序运行之前确定的，而我们所要输入的元素个数 n 是在程序运行时确定的，故不能保证所输入的元素个数不超出数组的范围。

然而，从逻辑上说，线性表和栈的元素个数（即长度）可以是一个非确定的有限整数，因此，采用这两种结构可以满足存储数据的要求。由于最后读入的元素要最先输出，符合“后

进先出”操作特点，故采用栈结构实现直观明了。为简化描述，下面不涉及栈的具体的存储结构形式，而只是采用算法调用的形式。

【算法描述】

```
void ReverseOrderOut()
{
    seqStack S;
    elementType x;
    int n;
    initStack(S);  //初始化栈，S.top=-1
    cout<<"请输入整数个数：n=";
    cin>>n;
    cout<<"请输入"<<n<<"个数据元素（整数）:"<<endl;
    for(int i=1;i<=n;i++)
    {
        cin>>x;
        pushStack(S,x);    //循环读入数据、入栈
    }
    cout<<"逆序输出：";
    while(!stackEmpty(S))
    {
        stackTop(S,x);    //取栈顶元素到 x
        cout<<x<<", ";    //输出栈顶元素值 x
        popStack(S,x);    //元素 x 出栈
    }
}
```

【例 3.2】设计算法将一个十进制整数转换为八进制数，并输出。

【解题分析】

数制转换的计算方法大家都很熟悉：循环相除取余数，对于本例即“除 8 取余”方法。设十进制数为 n，循环除 8，每次相除取出余数并保存，相除中商的整数部分再重新赋给 n，直至 n=0。

“除 8 取余”结束，最后获得的余数是对应八进制数的最高位，第一个获得的余数是最低位，即：余数要按获得次序的逆序（反序）输出，才得到八进制数。操作过程如图 3-4 所示。十进制数 1357 转换为八进制数为 2515。由分析可知，余数的输出次序正好要与获得次序相反，所以，利用栈的“后进先出”特性，可以很容易实现这个需求，即每取得一个余数，将其入栈，取余结束后，再依次将余数出栈打印即可。

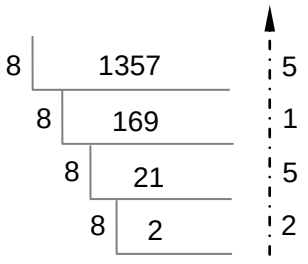


图 3-4 进制转换操作示意图

【算法描述】

```
void Dec2ocx(int n)
{
    seqStack S;
    int mod,x;
    initStack(S);   // 初始化顺序栈
    while(n!=0)
    {
        mod=n % 8;           //除 8 取余数到 mod
        pushStack(S, mod);   //余数入栈
        n=n/8;               //除 8 取商，存回 n
    }
    cout<<"八进制数为：";
    while(!stackEmpty(S))
    {
        popStack(S,x);       //取栈顶元素入 x，出栈
        cout<<x;             //打印余数
    }
}
```

【思考问题】十进制转换为二进制数呢？
十进制转换为十六进制数呢？
十进制转换为任意进制数呢？

【例 3.3】设计算法将单链表 L 就地逆置，即将链表中的各元素结点的后继指针倒置为指向其前驱结点，将 e_1 结点变成最后一个结点，e_n 结点变成第一个结点。

【解题分析】这一问题原本是第二章线性表中的问题，下面我们先用第二章单链表的知识求解此问题，再用栈的技术求解这个问题。就地逆置的含义是不要申请新结点，只能通过原链表的指针实现逆置。

（1）用单链表技术求解

方法是从原链表首元素结点开始，依次取出结点，采用头插法构建一个新表即可，只是这里不要申请新结点。这样原链表结点就被分为两个部分：已逆置部分、未逆置部分。未逆置部分用指针 L 指向第一个结点；已逆置部分用指针 P 指示第一个结点。逆置过程操作如图 3-5 所示：

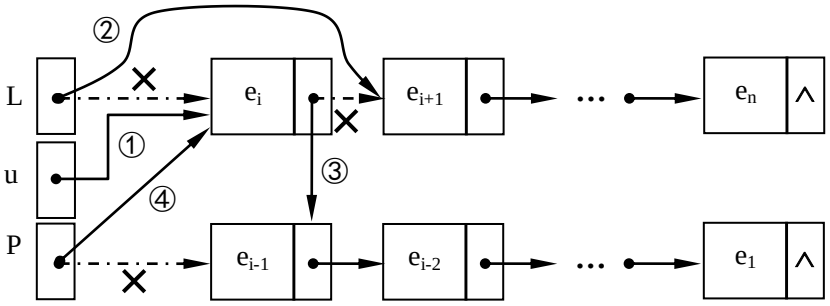


图 3-5 就地逆置过程示意图

由图 3-5 可见逆置的核心步骤有四步，图中用带圆圈的数字标注了四步操作的次序，同时用虚线表明了重新链接后废弃的指针，并在虚线旁加注了“×”号。

【算法描述】

```
void reverse(node* &L)
{
    node* P=NULL; //P 指向 e(i-1)节点
    node* u; //u 指向 ei 节点
    while(L!=NULL)
    {
        u=L;          //操作①，用指针 u 指示待分离的表头结点
                              //u 和 L 指向尚未逆置部分的第一个结点
        L=L->next;  //操作②，未逆置部分表头指针 L 后移指向 e(i+1)结点
        u->next=P;  //操作③，新分离出的 ei 结点的 next 指针指向 p，形成逆置
        P=u;          //操作④，已逆置部分头指针 P 指向新分离出的结点 ei
    }
    L=P; //原表头指针指向新的表头
}
```

（2）用栈技术求解

本题也可以利用栈的“后进先出”操作特点完成就地逆置操作，可以使用顺序栈，也可用链栈求解，下面给出使用顺序栈求解的算法描述。定义一个顺序栈，栈元素为单链表的结点指针，循环取出每个结点的指针，入栈。入栈后，原链表尾结点的指针存放在栈顶，首元素结点指针存放在栈底。构建逆置链表时，将栈内的指针依次弹出，按尾插法重建链表，这样就得到一个就地逆置的链表。栈的元素类型要定义为 typedef node* elementType。因为要采用尾插法，重建时要设置一个尾指针。下面的算法只处理不带头结点的单链表，如果单链表带有头结点，可在主调函数中处理，先产生不带头结点的链表头指针，再调用本函数。

【算法描述】

```
void reverse(node* &L)
{
    node *R, *u;
    seqStack S;
    initialStack(S);  //初始化栈
    u=L;
    while(u)
    {
        pushStack(S,u);  //链表中所有结点的指针入栈
        u=u->next;
    }
    if(stackEmpty(S))
        return;       //空栈，返回
    stackTop(S,u);    //取栈顶，即 en 的指针到 u
    L=u;              //设置新表头指针，即原表尾结点 en 的指针
```

```
    R=u;              //新表尾指针
    u->next=NULL;
    popStack(S,u);  //原表尾结点 en指针出栈
    while(!stackEmpty(S))
    {
        stackTop(S,u);     //取栈顶结点指针
        R->next=u;         //尾插法将当前结点链接到新表，形成逆置
        u->next=NULL;
        R=u;
        popStack(S,u);     //当前结点指针出栈
    }
}
```

【思考问题】本例中用顺序栈实现单链表的就地逆置，如何用链栈实现呢？

2. 表达式的计算

表达式计算指输入任意一个合法的表达式，计算机能自动计算出结果。是任何程序设计语言编译中的一个最基本问题，是栈应用的一个典型实例。

对这一问题，许多初学者可能会感到很奇怪，觉得表达式的计算对计算机来说是早已解决的基本问题了，用不着我们再费劲考虑了。然而，我们现在的问题不是要调用系统中求解表达式的程序，而是要自己编写程序来实现求解。

还有些初学者可能会考虑到对特定的表达式，用一段特定的程序来实现求解，这显然也不行，因这不满足前面所要求的“对任意输入的表达式的计算”的要求。

任何一个表达式都由操作数、运算符和定界符组成，**操作数**指参与计算的数值，可以是常量和变量，以及中间计算结果；**运算符**用来对操作数进行不同的运算，分为算术运算符、关系运算符和逻辑运算符三类；**定界符**是用来改变计算次序的特殊符号对，它们成对出现，以界定左右边界，常见的定界符为成对出现的括号。

运算符有优先级之分，用来确定计算的先后次序，为简单起见这里我们只讨论算术运算符。算术运算符优先级规定与数学相同—先乘除，后加减；顺序出现的同级运算符，先出现的优先级高。定界符可以改变运算次序，规定：一对定界符外部任何其它运算符的优先级低于定界符；一对定界符内部所有运算符的优先级高于定界符。

通用表达式的求解是软件设计领域中的重要成果之一，需要借助于栈这种数据结构来实现，基本求解思想如下：

（1）设置两个栈分别存储表达式中的操作数和运算符，不妨称之为**操作数栈**和**运算符栈**。这里操作数也可能是中间计算结果。定界符视作算符，入运算符栈。

（2）求解时，依次扫描表达式中的各基本符号（每个运算符、操作数等均看作是一个基本符号），并根据所扫描的符号的内容分别做如下处理：

①如果所扫描的基本符号是操作数，则将此操作数直接进到操作数栈中，然后继续扫描其后续符号。

②如果所扫描的基本符号是运算符 CurrentS，则以此来决定运算符栈当前栈顶的运算符 TopS 是否能运算，具体地说，分情况处理如下：

——如果栈顶的运算符 TopS 的优先级低于当前所扫描的运算符 CurrentS 的优

先级，则栈顶的运算符不能进行运算，故当前运算符 CurrentS 要入栈，然后继续扫描其后续符号。

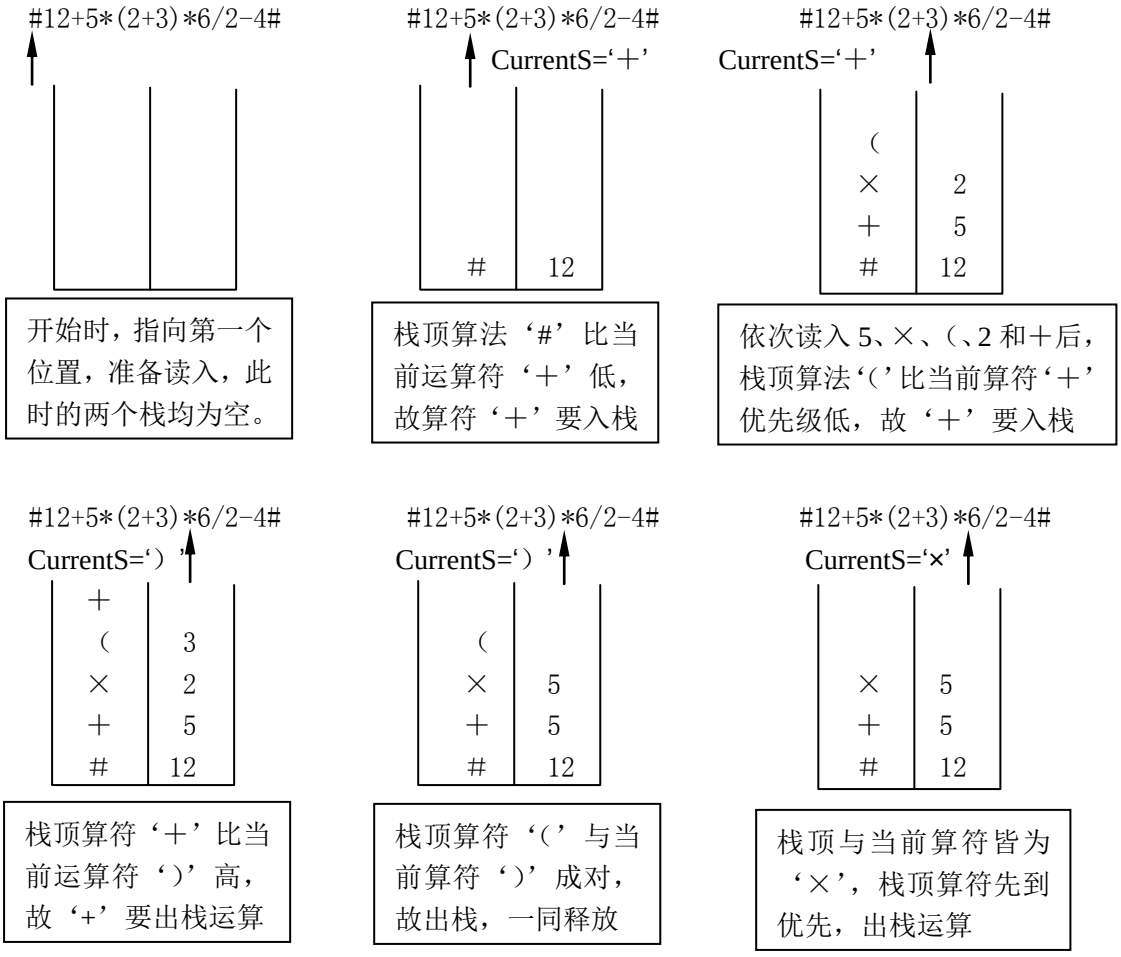
——如果栈顶的运算符 TopS 的优先级高于当前所扫描的运算符 CurrentS 的优先级，则要取出运算符 TopS 进行运算。此时，TopS 的两个操作数一定是操作数栈栈顶位置的两个元素，取出这两个元素进行计算，并将运算结果入操作数栈。

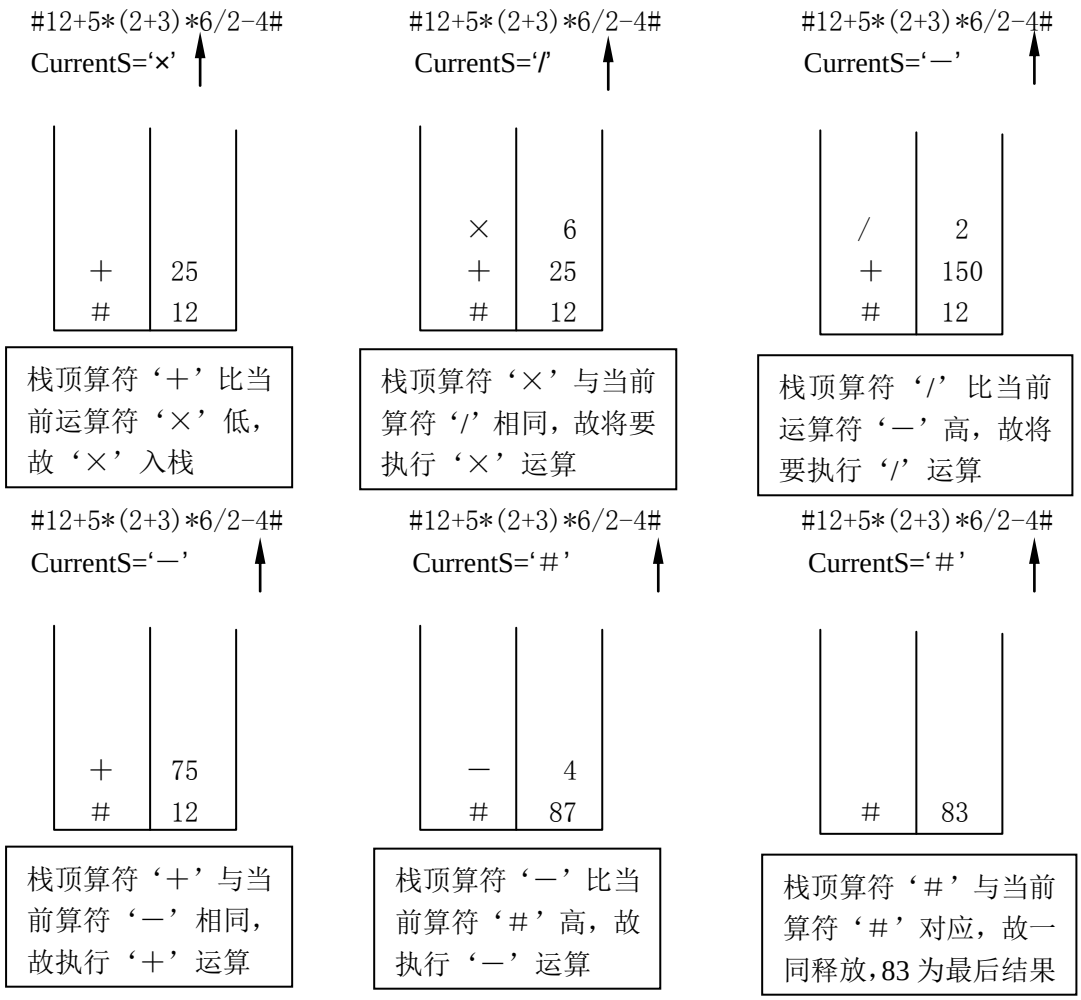
如果此时的运算符栈不空，需要继续以栈顶的运算符 Tops 对当前运算符 CurrentS 重复上述比较操作，以此确定是用栈顶运算符运算还是将当前运算符入栈。

当表达式扫描结束，即计算出了表达式的结果。

【例 3.4】设计算法实现对任意输入的表达式的计算。例如表达式为 12+5*(2+3)*6/2-4。

【解题分析】下面以表达式 12+5*(2+3)*6/2-4 为例来演示求解过程中的栈的变化情况。为便于描述，在表达式首尾添加一对定界符“#”，即在表达式首尾分别添加一个“#”，表示表达式的开始和结束，并将运算符栈和操作数栈两个栈并列画在一起。扫描过程中用一个箭头表示将要读入的位置。





表达式求解算法较为复杂，这里不在给出具体算法描述，有兴趣的读者可参阅有关资料自行完成。

3.2 队列

队列是模仿人类“排队”法则构建出来的一种数据结构，人类在争用某些稀缺资源时，为体现公平，往往采取排队的解决办法，按“先来先服务”的原则使用这些资源。提到排队，首先浮现在脑海的可能是每年春运数以亿计的人排队购买车票、机票，排队等待上火车的景象，场面壮观而混乱，参杂着欢乐、劳累和痛苦，给每个置身其中的人留下深刻的印象。此外，对学生来说，到食堂排队买饭是差不多每天发生的事情。

计算机系统中队列的使用更是比比皆是，早期的主机加终端系统中，多个终端为争用主机 CPU 使用权，就采用排队的解决方法；很多 CPU 的内部都带有取指令队列，以加快指令的执行速度；计算机主机向外设传输数据，比如打印机打印文档，当外设速度较慢，来不及处理主机传来的数据时要使用队列对数据进行缓冲；操作系统中的进程、线程调度等用到队列，Windows 操作系统更是大量使用信息队列，利用消息驱动来调度和管理计算机系统，系统运行时构建众多的消息队列，将不同的消息放入不同的消息队列进行管理；网络应用系统中队

列也得到广泛的使用，比如 C/S 模式系统中，服务器为了处理众多客户端的并发访问，会用队列对客户端的服务请求进行管理；此外，解决实际问题的应用系统中也会经常使用队列。可见，队列也是一种重要的、应用广泛的基础数据结构。

3.2.1 队列的定义和运算

1. **基本概念**

队列（Queue）是限定只能在一端插入、另一端删除的线性表。允许删除的一端叫做**队头**（front），允许插入的一端叫做**队尾**（rear），如图 3-6 所示。称没有元素的队列为**空队列**。由定义知，队列也是操作（运算）受限的线性表。

我们称队列的插入操作叫**入队**，队列的删除操作叫**出队**。

队列**特性**的讨论：假设初值为空的队列，按 $e_1, e_2, \cdots, e_n$ 次序，将这些元素依次连续入队，由定义可知，这些元素的出队次序只能是 $e_1, e_2, \cdots, e_n$，也就是说，队列具有**先进先出**(FIFO--First in,first out)的特性。

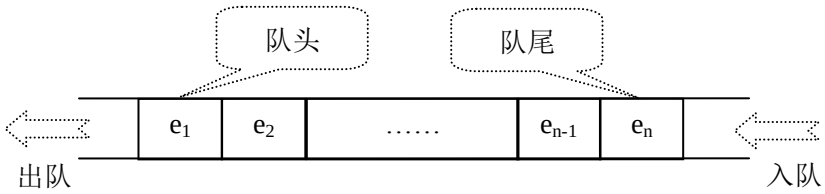


图 3-6 队列结构示意图

2. **队列的运算**

与栈类似，队列有以下几个常用的基本运算：

(1)初始化队列：initialQueue(Q)

设置队列 Q 为空。

(2)判队列是否为空：queueEmpty(Q)

判定队列是否为空。若队列 Q 为空，返回 TRUE，否则返回 FALSE。

(3)取队头元素：queueFront(Q, x)

若队列 Q 不空，求出队列 Q 的队头元素置 x 中，否则，提示取队头出错。

(4)入队：EnQueue(Q,x)

将值为 x 的元素插入到队列 Q 中。若插入前队列已满，不能入队，报出错信息。

(5)出队：OutQueue(Q,x)

若队列 Q 不空，删除队头，并将该元素的值置 x 中，否则，报出错信息。

(6)判队列是否为满：queueFull(Q)

若 Q 为满时，返回 TRUE，否则，返回 FALSE。这一运算只用于顺序队列，链队不存在队满问题。

下面分别讨论队列的两种存储结构及其运算实现。

3.2.2 顺序队列与循环队列

1. **存储结构**

和顺序表一样，以顺序存储方式存储的队列叫做**顺序队列**（Sequential Queue）。也可

用一个数组 data[MAXLEN]来存储元素，将数组前面的元素作为**队头**，后面的元素作为**队尾**，并分设两个整型变量 **front** 和 **rear** 来指示队头和队尾元素，称为**队头指针**和**队尾指针**（它们并非真正的指针，而只是整型变量）。

读到这里有的读者可能会有一个疑问？既然用数组存放队列元素，能不能固定以 data[0]为队头，只设一个尾指针 rear 呢？干嘛还要设置一个队头指针 front 呢？我们简单分析一下，就会明白其中的道理。如果只设队尾指针，入队操作非常简单，不需移动元素，直接把元素插入到队列后面即可。但是，出队操作时，每次要删除 data[0]的数据，由前面学习的顺序表知识可知，完成这个操作需要把队列中除 data[0]以外的所有元素前移一个单元，这是一个非常耗时的操作。此外，每次出队还需要修改队尾指针 rear。

使用队列通常是因为某种资源紧缺，发生争用，我们当然希望队列的操作是高效的，显然，上面讨论出现的批量移动元素是不可取的。为此，我们在队列中增加一个 front 指针，这样，入队操作不变，出队操作时，我们只需简单的执行 front++，即队头指针后移一个单元即可，无需批量迁移元素。

关于 front 指针和 rear 指针的具体指向，不同的材料有不同的处理。一种处理是让 front 指示**队头元素的前一个位置**，而不是指在队头元素上；rear 指示队尾元素。另一种处理是 front 指示队头元素，而 rear 指向队尾元素的后一个位置。为什么这样做呢？这样错开指示对普通的顺序队列并没有什么用处，但在稍后介绍的循环顺序队列中，可谓意义重大，它让我们能够区分出队满和队空两种状态。本书采用上述第一种处理方式。由此我们得到顺序队列的结构如图 3-7 所示。

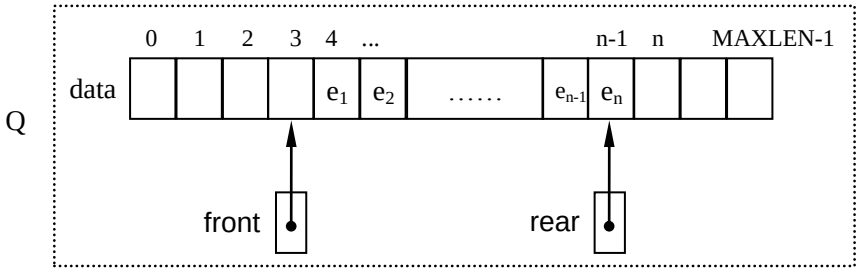


图 3-7 顺序队列结构示意图

【顺序队列存储结构描述】

```
typedef struct sQueue
{
    elementType data[MAXLEN];   //存放队列元素
    int front;      //队头指针
    int rear;       //队尾指针
}seqQueue;
```

2．普通顺序队列存在的问题

在这样的普通顺序队列中，入队操作就是先将尾指针 rear 后移一个单元（rear++），然后将元素值赋给 rear 单元（data[rear]=x）。出队时，则是头指针 front 后移（front++）。象这样进行了一定数量的入队和出队操作后，可能会出现这样的情况：尾指针 rear 已指到数组的最后一个元素，即 rear==MAXLEN-1，此时若再执行入队操作，便会出现队满“溢出”。

然而，由于在此之前可能也执行了若干次出队操作，因而数组的前面部分可能还有很多闲置的元素空间，即这种溢出并非是真的没有可用的存储空间，故称这种溢出现象为“**假溢出**”。显然，必须要解决这一假溢出的问题，否则顺序队列就没有太多使用价值。下面介绍的循环顺序队列就巧妙地解决了这个问题。

3．循环顺序队列

循环顺序队列的存储结构，头、尾指针都和普通顺序队列相同。不同的只是我们将数组 data[]视为“环状结构”，即视 data[0]为紧接着 data[MAXLEN-1]的单元，为相邻单元，首尾相接构成一个“环”。这样我们得到循环顺序队列的结构如图 3-8 所示。

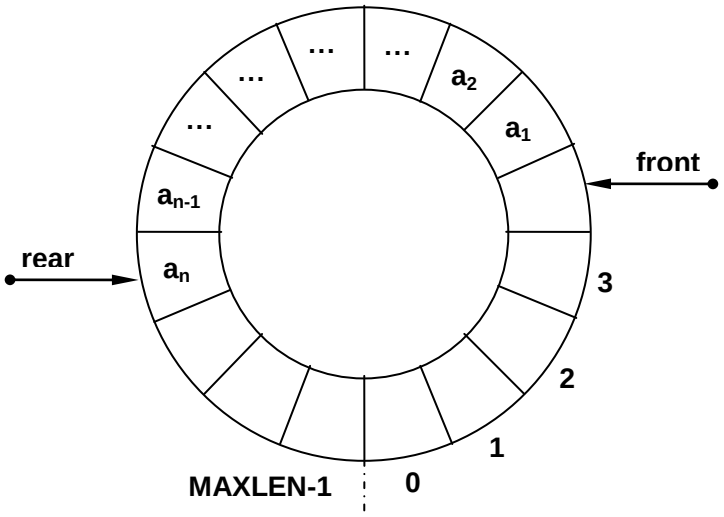


图 3-8 循环顺序队列结构示意图

有图 3-8 可见，如果 rear 当前指向数组 data[]的最后一个单元，即 rear==MAXLEN-1 时，欲再插入一个元素，执行 rear++，则 rear==MAXLEN，对普通顺序队列，此时 rear 已经指向 data[]数组以外，一定会“溢出”。但对循环队列，rear++后，我们让 rear==0，即重新指向 data[0]单元，如果此时 data[0]单元为空，则仍然可以完成插入操作。对 front 指针，我们也一样处理，当 front==MAXLEN-1 时，再删除一个元素，front++后，让 front==0，重新指向 data[0]单元。通过上述规定，我们产生了一个逻辑上的循环结构（非物理上的），data[]数组中的所有单元都可以循环重复使用，不会出现“假溢出”问题。

因为反复的入队和出队操作，循环队列中，front 和 rear 指针可能位于数组 data[]的任何位置，如图 3-8 所示。可能出现 front>rear 的情况。

对循环队列我们如何计算入队（插入）操作 rear 指针的位置呢？正常情况下，rear<MAXLEN-1，则 rear++就是插入（入队）位置。但当 rear==MAXLEN-1 时插入，执行 rear++后，不能让 rear==MAXLEN，而要让 rear==0。对此，我们可用下述两种方法实现：

【方法一】手工判断

```
if( rear==MAXLEN-1)
    rear=0;
else
    rear++;
```

在 C 语言中，这种方法可用一句代码表示：

rear=(rear+1==MaxLen) ? 0 : rear ++ ;

【方法二】模运算

rear=(rear+1) % MAXLEN;

模运算几乎是每种高级程序设计都支持的运算，利用模运算可以很容易实现一个逻辑上的环。举个具体的例子，比如 MAXLEN=100，当 rear=99 时，再插入元素，rear+1 后，rear 为 100，对其做模运算求余数，即 100 % 100 == 0，可见余数正好为 0，将其赋给 rear，正好就是插入位置，实现了逻辑循环。在 rear<99 情况下，比如 rear=67，插入元素，rear+1 后为 68，对其做 100 的模运算，仍为 68，也是插入位置。综上分析，无论 rear 原先指在什么位置，插入操作时，用模运算都可以计算出最终目标位置。显然这种方法比前一种方法简洁。

插入操作位置计算问题解决了，那么出队（删除）操作如何计算位置呢？回答是：计算方式与删除操作一样，也有上述两种方法，比如，用方法二模运算实现，代码如下：

front=(front+1) % MAXLEN;

还有一个问题是怎样判断循环队列队空和队满呢？如果按常规做法，让 front 指向队头元素，rear 指向队尾元素，在队空和队满时，都会出现 front==rear，这样就无法分辨出究竟是队空还是队满。解决这个问题也有两种方法如下：

【方法一】data[]数组中保留一个单元空间

这个方法我们在前面已经提到过，即在队列已经有元素情况下（队列非空），让 front 指向队头元素的前一个单元（不指在元素上，而指向一个空的位置），rear 指向队尾元素，如图 3-8 所示。即 front 指向单元始终为保留单元，不存放元素，长度为 MAXLEN 的 data[]数组，最多只存放 MAXLEN-1 个元素。这样，无论怎样执行入队、出队操作，只要队列中有元素，rear 就永远不会“赶上”front。在队列满时也是这样，front 会指在 data[]数组剩下的唯一的一个空单元上，rear 指在队尾元素上，front 和 rear“相差 1 个单元”（相差一个单元是循环意义上的）。队列空时，我们让 front==rear。这样就可以区分出队空和队满情况。下面用一个实例来说明循环队列正常、队空和队满的情形，如图 3-9 所示。

图 3-9（a）表示一个正常的非空队列，front=2，rear=5；图 3-9（b）表示一种队空情形，front=rear=4；图 3-9（c）表示一种队满情形，front=2，rear=1。

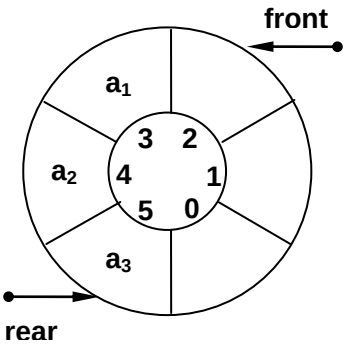
可见，循环顺序队列中，无论哪种情况，front 和 rear 指针都可能指在 data[]数组的任何单元上。可能有 front<rear，也可能 front>rear。队空时 front 不是固定指向 data[0]单元，可能指向 data[]的任何位置，只要 front==rear。队满时 rear 不是固定指向 data[MAXLEN-1]单元，可能指向 data[]的任何位置，只要 front==(rear+1) % MAXLEN。由此分析我们可以得出循环顺序队列判定队空和队满的条件：

【队空判定条件】front==rear

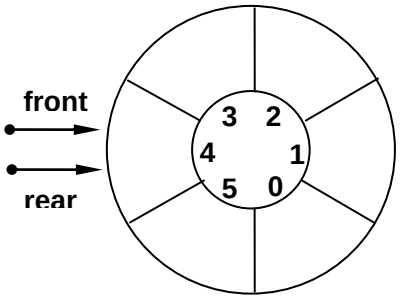
【队满判定条件】front=(rear+1) % MAXLEN

为什么队满判定条件要做模运算呢？正常情况 rear<MAXLEN-1 时，直接用 front==rear+1 判定即可。但有一种特殊情况的队满，front=0，rear=MAXLEN-1，这种情况必须模运算后才能判定。

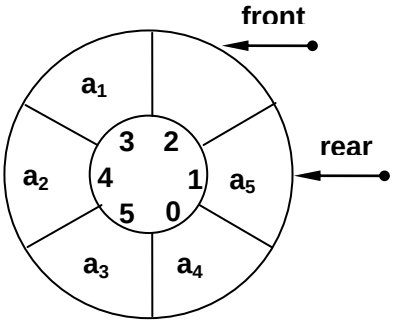
本书即采用这种方法判定队空和队满。



（a）正常情况的循环队列



（b）循环队列队空示意图



（c）循环队列队满示意图

图 3-9 循环队列队空、队满示意图

【方法二】设置入队和出队标志

设置一个标志变量 s，用来区分入队、出队操作，比如，设变量为 s，s==1 标记插入操作，s==0 标记删除操作。在队列空和满时，都有 front==rear，但队满只会在执行插入操作后发生；队空只会在初始化和执行删除操作时发生。所以(front==rear) && (s==0)为队空；(front==rear) && (s==1)为队满。这种方法相对繁琐，本书采用方法一判定队空和队满。

4．循环顺序队列基本运算实现

(1)初始化队列：

由前面的讨论可知，循环队列为空时，其头尾指针相等，且约定同时指向 data[0]单元。

【算法描述】

```
void initialQueue( seqQueue *Q )
{
    Q->front=0;  //空队列
    Q->rear=0;
}
```

(2)判断队空

即判断头、尾指针是否相等。

【算法描述】

```
bool queueEmpty(seqQueue &Q)
{
    if(Q.front==Q.rear)
        return true;   //队空，返回 true
    else
```

```
        return false;  //队不空，返回 false
}
```

(3)判断队满

按前面的约定，队列满时，front 指向唯一的保留单元。rear 的下一个单元即 front（循环意义上的），判定条件前面已经给出。由此得算法如下：

【算法描述】

```
bool queueFull(seqQueue &Q)
{
    if(((Q.rear+1) % MaxLen)==Q.front)
        return true;  //队满，返回 true。即尾指针的下一个位置是头指针。
    else
        return false;  //不满，返回 false
}
```

(4)取队头元素

先要判定队列是否为空，队空给出错误信息，队列非空取出队头元素。

【算法描述】

```
void queueFront(seqQueue &Q, elementType &x)
{
    if(queueEmpty(Q))
        cout<<"队空，不能取队头元素！"<<endl;
    else
        x=Q.data[(Q.front+1) % MaxLen];  //front 指示的下一个单元才是队头元素
}
```

【思考问题】x=Q.data[(Q.front+1) % MaxLen]中，为何要做模运算呢？

(5)入队

入队前，首先要判断是否已经队满，若队满，则不能插入，报错。

【算法描述】

```
void enQueue(seqQueue* Q, elementType x)
{
    if(queueFull(*Q))
        cout<<"队列已满，不能完成入队操作！"<<endl;
    else
    {
        Q->rear=((Q->rear)+1) % MAXLEN;  //后移 rear
        Q->data[Q->rear]=x;              //填入数据 x
    }
}
```

(6)出队

出队之前先要判断是否队空，队空，删除失败，报错。对出队的队头元素，也可以用参数返回主调函数，但是这里给出的算法是简单的删除函数。

【算法描述】

```
void outQueue(seqQueue* Q)
{
    if(queueEmpty(*Q))
        cout<<"空队列，没有元素可供出队！"<<endl;
    else
    {
        Q->front=(Q->front+1) % MaxLen;  // front 指针后移一个单元
    }
}
```

【思考问题】如果使用函数参数传回删除的队头元素，上述算法要如何修改？

【算法分析】与栈的基本运算类似，队列的六个基本运算的时间复杂度都是 O(1)。

3.2.3 链队列

顺序队列或者循环顺序队列都只适合于事先知道其最大存储规模的情况。然而，如果事先不能估计队列的最大存储规模，则需要采用能动态申请内存的链式存储结构来存储队列，由此得到**链队列**。

1．链队列的存储结构

在采用链队列存储时，要确定结点结构，队头和队尾位置等问题，讨论如下：

（1） 链队列采用单链表结构，结点结构与单链表相同；

（2） 要区分队头和队尾，我们约定用单链表的表头最为队头，并设置队头指针 front；单链表的表尾最为队尾，并设置队尾指针 rear。

（3） 为操作方便我们也给链队列添加一个头结点，形成带头结点的链队列，这样队头指针指向头结点，而 front->next 才指向队头元素结点；rear 仍指向队尾结点。

由此，得到链队列建构如图 3-10 和图 3-11 所示。

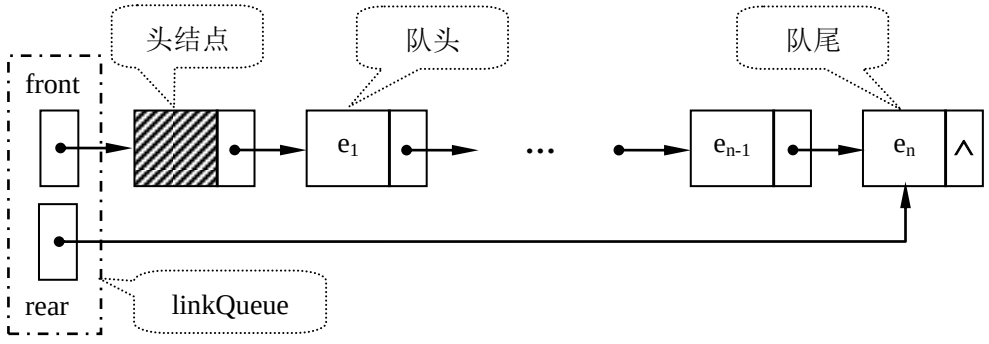


图 3-10 链队列结构示意图

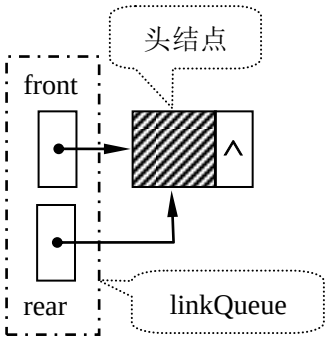


图 3-11 空链队列示意图

由上面的讨论和图 3-10，图 3-11 可见，**front 和 rear 指针唯一地确定一个链队列**。但是出现了一个新的问题，描述一个链队列需要 front 和 rear 两个指针，而这两个指针又不是每个结点都有的，所以不能像单链表和链栈那样只定义结点结构就行了。怎么处理呢？我们采用两步定义方式，先定义链队列的结点结构，结点结构定义同单链表；再定义一个结构描述 front 和 rear 指针，这个结构表示一个队列。

【链队列结点结构描述】

```
typedef struct LNode
{
    elementType  data;   //存放数据元素
    struct LNode *next;  //下一个结点指针
} node;
```

【链队列结构描述】

```
typedef struct
{
    node *front;   //队头指针
    node *rear;    //队尾指针
}linkQueue;
```

因为 front 和 rear 指针唯一确定一个链队列，所以我们可以用 linkQueue 来定义一个链队列。

【思考问题】

①front、rear 和 next 三种指针类型相同吗？

②如果 front、rear 和 next 指针类型相同，为什么分两步定义呢？

2．链队列的基本运算实现

(1)链队列初始化

初始化链队列即建立一个空队列，如图 3-11 所示，需要申请一个新结点（头结点），next 置为 NULL，front 和 rear 同时指向头结点。同时注意初始化后的链队列，向主调函数回传的实现方法。

【算法描述】

```
void initQueue(linkQueue &Q)
```

```
{
    Q.front=new node;     //产生头结点，指针为 front;
    Q.rear=Q.front;       //rear 也指向头结点
    Q.front->next=NULL;   //头结点的 next 置为 NULL
}
```

(2)判断队空

有两种方法可以判断链队列是否为空，一种方法是 Q.front->next==NULL；另一种方法是判断其头尾指针是否相同，即 Q.front==Q.rear。这里采用第二种方法。

【算法描述】

```
bool queueEmpty(linkQueue &Q)
{
    return (Q.front==Q.rear);
}
```

(3)取队头元素

先判断是否队空，队空的话，给出相应信息。本算法取出的队头用参数返回给主调函数。

【算法描述】

```
void queueFront(linkQueue &Q, elementType &x)
{
    if( queueEmpty(Q) )
        cout<<"空队列，无法取队头元素！"<<endl;
    else
        x=((Q.front)->next)->data;  //队头是 front->next 指向的结点
}
```

(4)入队

将值为 x 的元素插入队列，申请新结点，新结点赋值 x，插入到队尾。

```
void enQueue(linkQueue &Q, elementType x)
{
    node* P=new node;   //申请内存，产生新结点
    P->data=x;          //x 赋给新结点
    P->next=NULL;       //x 结点成为新的尾结点，next 置 NULL
    Q.rear->next=P;     //新结点链接到原表尾
    Q.rear=P;           //后移尾指针，指向新结点（新队尾）
}
```

(5)出队

首先判定是否队空，队空时给出错误信息。然后删除队头结点，也可以在删除队头的同时用参数传回删除的队头元素值，下面的算法就用参数 x 传回删除的队头元素值。

【算法描述】

```
void outQueue(linkQueue &Q, elementType &x)
{
    node* u;   //用以指向删除节点
    if(QueueEmpty(Q))
```

```
        cout<<"当前队空，无法执行出队操作！"<<endl;
    else
    {
        x=Q.front->next->data;    //取出队头元素值到变量 x
        u=Q.front->next;          //u 指向队头（首元素结点）
        Q.front->next=u->next;    //更新队头指针
        delete u;                 //删除原队头，释放内存
        if(Q.front->next==NULL)   //如果删除结点后，队空，则要修改 rear 指针
            Q.rear=Q.front;
    }
}
```

特别需要注意算法最后的更新 rear 指针代码，当队列只有一个元素结点时，队尾指针就指在这个结点上，删除这个结点后，将成为空队列，如果不更新 rear 指针，rear 将指向不确定的内存位置，故要更新 rear，使 rear=front，如图 3-12 所示。

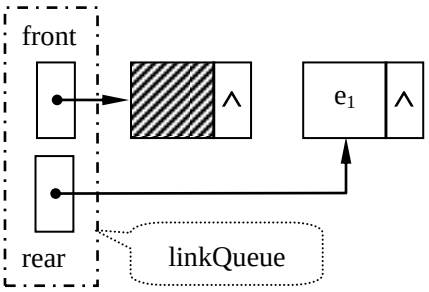


图 3-12 删除链队最后一个元素结点中间状态示意图

队列在软件设计中有较多的应用，最典型的应用是在后面要介绍的图的广度遍历算法中。

3.3 栈与递归

递归（Recursion）是我们思考问题和分析问题的一种方式，是解决问题的一种方法。计算机中实现递归计算需要借助栈，是栈的另一个重要应用。事实上，计算机中的函数调用也是用栈来实现的；将递归程序转换为非递归程序也经常要借助栈来实现。

递归是问题归约法（Problem Reduction）或分治法求解问题的一种。人类在解决复杂问题时，往往采用“分而治之”的策略，即将原问题进行反复分解，分解为一些规模较小的子问题，直到产生的所有子问题都可以直接求解，这些可以直接求解的子问题叫做基本问题（Primitive Problem）。然后，通过解决所有基本问题，反向合成出原始问题的解。递归求解是上述情况的一个特例，也分为递推分解子问题和回推原问题的解两个过程，但要求递推分解的子问题与原问题有相同的定义及相同的求解方法。

计算机求解递归问题时，通常将问题的定义、分解、求解用函数或过程的形式来实现，递推分解子问题时就会出现函数调用自身的情况，我们称函数直接或简介调用自身叫做递归。

3.3.1 递归的基本概念

1. 递归的定义

如果一个对象部分地包含它自己，或者利用自己定义自己的方式来定义或表述， 则称这个对象是递归的；如果一个过程（函数）直接或间接地调用自己，则称这个过程（函数）是一个**递归过程（函数）**。

若在函数体内直接调用自己，称为**直接递归**；若函数调用其它函数，其它函数中又反过来调用前者，称为**间接递归**。

我们在先修的一些课程中已经接触过递归函数，这里给出几个递归函数的实例。

(1)求整数的阶乘 n!，阶乘的递归定义如下：

$$n!=\begin{cases}1 & n=0\\ n\times(n-1)! & n>0\end{cases}$$

【算法描述】

```
int Fact( int n )
{
    if(n==0)
        return 1;              //终止条件
    else
        return n*Fact(n-1);  //递归调用
}
```

(2)求整数的 Fibonacci 数，Fibonacci 数递归定义如下：

$$Fib(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ Fib(n-1)+Fib(n-2) & n\geq 2\end{cases}$$

【算法描述】

```
int Fib(int n)
{
    if( n==0 )
       return 0;   //终止条件
    else if(n==1)
       return 1;   //终止条件
    else
       return Fib(n-1)+Fib(n-2);  //递归调用
}
```

(3)函数中两次调用自身

【算法描述】

```
void P(int n)
{
      if(n>0)    //终止条件：n<=0
```

```
    {
        P(n-1);     //递归调用
        cout<<n;
        P(n-2);     //递归调用
    }
}
```

(4)间接递归调用

【算法描述】

```
void P1(int n)
{
    if(n>0)
    {
        cout<<n;
        P2(n-1);
    }
}
```

```
void P2(int n)
{
    if(n>0)
    {
        P1(n-1);
        cout<<n;
    }
}
```

2.递归调用需要具备的条件

（1）待解问题可以反复分解为子问题，子问题的求解复杂度趋于简单，且子问题和原问题具有相同的定义和求解方法；

（2）正向递推分解子问题的过程必须有明确的结束条件，叫做递归终止条件，或**递归出口**。即把问题反复分解，直至产生一个可以直接求解的基本问题集合，正向递推结束。

如果没有明确的递归出口，正向分解过程将一直进行下去，对计算机程序来说，这相当于“死循环”，直至栈溢出错或系统崩溃。

（3）通过基本问题的解可以反向回归合成出原问题的解。

可见，递归求解分为两个过程，一个是正向递推分解子问题的过程，另一个是通过基本问题的解反向回归合成原问题解的过程。

3.递归函数的一般形式

由上面的实例和讨论，我们可以得到递归函数的一般形式如下：

```
void  RecFunc(参数表)
{
    if(递归出口条件)
        简单操作;     //递归出口（终止条件）
    else
    {
        简单操作;
        RecFunc(参数表);        //递归调用
        [简单操作;]             //加[]表示可选项。下同。
        [RecFunc(参数表);]      //可能有多次递归调用
        [简单操作;]
    }
```

}

这里给出的函数形式没有返回值，但有的递归函数是需要返回值的，且返回值还可能参与运算，例如前面给出的实例（1）求阶乘和实例（2）求 Fibonacci 数。

3.3.2 递归调用的内部实现原理

针对许多初学者理解递归存在困难，我们接下来介绍递归的实现原理。事实上递归函数调用也是函数调用，只不过是调用函数自身，其实现原理与普通函数调用实现原理是一样的。所以要搞清楚递归实现原理，首先要弄明白一般函数调用的原理。

1.一般函数调用的实现原理

（1）函数调用要解决的问题

从上面的场景中我们看以看出，函数调用需要解决如下问题（为了表达方便下面我们称被调用函数为子函数）：

①如何找到子函数（过程）的入口

我们知道每个函数都有函数名，编译成机器代码后，函数名就变为子函数第一条指令的存储地址，当调用此子函数时，比如上面的 call B，CPU 的程序指针指向这个地址（函数名），就找到了子函数的入口。

②如何向子函数传递参数

许多情况下函数带有参数，比如前面给出的实例（1）求阶乘和实例（2）求 Fibonacci 数，执行函数调用时，需要主调函数向子函数传递这些参数（实参），这个工作是用栈来完成，即在转入子函数执行之前，把实参入栈，详情见稍后的讨论。

③子函数执行完毕如何找到返回位置（返回地址）

子函数执行完毕，应该返回到主调函数继续执行，即返回到主调函数调用位置后面的第一条指令继续执行，。

所以在转入子函数执行之前，需要把主调函数调用指令后面第一条指令的地址“告知”子函数，以便子函数执行结束能正确返回，这个工作也是通过将返回地址入栈完成的。

④子函数本地变量存储问题

子函数的本地变量也入栈保存，这样做对递归调用特别有用，参见下面的讨论。

⑤函数返回值问题

子函数可以通过函数参数和函数返回值两种方式往主调函数回传数据。数据回传是通过在主调函数和子函数间共享内存实现的。

函数参数回传数据时，先要在主调函数中定义一个变量，比如 x，为变量 x 在内存中开辟一块存储空间，然后把变量 x 的地址传递到子函数，子函数通过 x 的地址操作这块存储空间改变 x 的值，主调函数也是操作这块空间改变 x 的值。这样主调函数和子函数改变 x 的值，操作的是同一块内存空间，所以子函数中改变了变量 x 的值，主调函数是能够“感知”到的，从而实现了数据的回传，C++中指针和引用回传数据就是这样实现的。

函数返回值回传数据情况稍微复杂一些，对于字节数较少的返回值，直接通过 CPU 的寄存器回传，即子函数返回时，把要返回的值存放到 CPU 的寄存器中，返回后主调函数直接到 CPU 指定寄存器中读取这个值即可。比如对于 32 位机，4 字节以下的返回值，用单个寄存器返回；5-8 字节的返回值用 2 个寄存器返回。

对于较大的返回值，即长度超出 2 个寄存器长度的返回值，也是转换为参数进行传递的，只是参数是隐藏的。这个工作由编译器自动完成，编译器根据函数返回值类型，计算出需要的存储空间，如果超出 2 个寄存器长度，它就自动定义一个隐藏变量，开辟一块内存，调用子函数之前，把隐藏变量的地址传递给子函数，与②中讨论的实参传递一并完成，就是主调函数向子函数传递实参时，多了一个作为函数返回值的隐藏变量，可见这就是参数回传。

为了简化讨论，我们假设有一个“回传变量”，通过此回传变量把函数返回值从子函数返回到主调函数。

2.函数调用栈

操作系统执行程序时，为每个线程的函数调用专门开辟一块内存空间，以栈的方式进行管理，这块空间叫做**函数调用栈（Call Stack）**，也叫函数工作站。函数调用栈是线程独立的，即每个线程有自己独立的调用栈。

每当调用一个函数时，系统就在调用栈中为设个函数划出一片区域，存储这个函数调用的相关信息，这个区域叫做**栈帧（Stack Frame）**。栈帧组成如图 3-13 所示。栈帧中保存有以下信息：

函数实参 -- 主调函数传递给子函数的实际参数。如果函数有较大返回值，需用隐藏参数回传时，隐藏参数也作为实参保存在这里。

返回地址 -- 主调函数调用点后下一条指令的地址。

相关寄存器的值-- CPU 的一些寄存器值，细节不在这里介绍。

子函数的局部变量

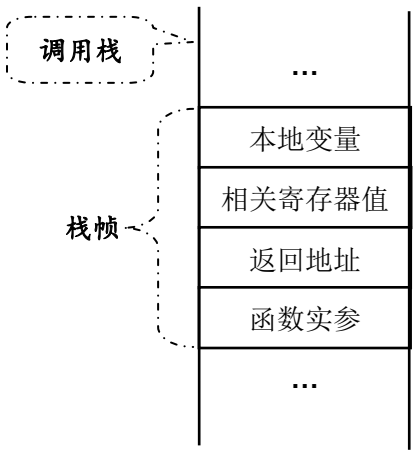


图 3-13 栈帧构成示意图

调用函数时，相关信息入栈，形成栈帧；子函数执行结束，相关信息出栈，对应的栈帧释放。

当函数有嵌套调用时，每调用一次函数就形成一个栈帧，调用栈中就会同时存在多个栈帧。按栈的先进后出原则，当前正在执行的函数的栈帧处于栈顶位置，叫做**活动栈帧**，也叫**活动记录（AR—Active Record）**。

下面用一个实例说明栈帧的工作情形，假定在主函数 main()中调用子函数 A()，A()中再调用 B()。三个函数代码如下，main()函数中符号“①”作为调用函数 A()返回后继续执行的地址，因为 A()返回后要执行 m=A(m)的赋值语句，我们设 A()返回后继续执行的地址就是赋值语句的地址，即“①”。同样，函数 A()中符号“②”表示调用 B()的返回地址。在栈帧中就直接用这两个符号表示返回地址；main()函数返回地址由系统确定，栈帧中用“main 返回地址”表示。图 3-14 和图 3-15 用来说明，程序执行时调用栈和栈帧的建立和释放过程。

【主函数代码】

```
int main()
{
        int m=5;
①:     m=A(m);
        cout<<m;
        return 0;
}
```

【A 函数代码】

```
int A(int n)
{
       int x=10;
②:    x=B(x+n);
       return x+n;
}
```

【B 函数代码】

```
int B(int w)
{
     int y=50;
     return y+w;
}
```

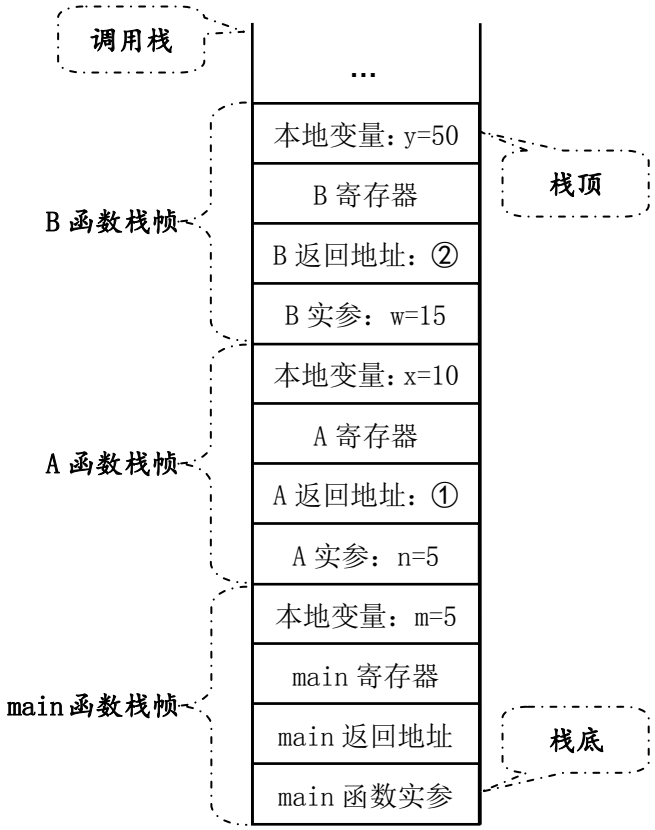


图 3-14 嵌套调用调用栈示意图

下面解释函数时调用栈和栈帧的建立和释放过程：

（1）启动程序，调用程序入口的主函数 main()（C、C++、Java、C#等语言程序都以 main()函数为程序驱动入口），执行 main()函数也是函数调用，相关信息压入调用栈，形成 main()函数的栈帧。如图 3-14 和图 3-15(a)所示。

（2）main()函数执行到①处，调用子函数 A()，先将实参（值为 5）入栈；再将 A()的返回地址“①”入栈；相关寄存器值入栈；最后将 A()的本地变量 x（值为 10）入栈，形成 A()的栈帧，然后即转入执行函数 A()。A()的栈帧处于栈顶位置，如图 3-14 和图 3-15(b)。

（3）函数 A()执行到“②”处，又调用子函数 B()，首先将 A()传递给 B()的实参（值 15）入栈；B()的返回地址“②”入栈；相关寄存器值入栈；B()的本地变量 y（值为 50）入栈，形成 B()的栈帧，然后转入函数 B()的执行。B()的栈帧处于栈顶位置，如图 3-14 和图 3-15(c)。

在函数 B()执行期间，调用栈中共有 3 个栈帧，其中函数 B()的栈帧为活动记录，处于栈顶位置，表示当前正在执行函数 B()的代码，如图 3-14 所示。至此正向调用过程结束。

（4）当函数 B()执行结束，B()函数栈帧中本地变量首先出栈；接着 B()相关寄存器出栈；B()的返回地址出栈，并取得返回地址“②”；B()实参出栈。至此 B()的栈帧销毁，B()函数返回值（值为 65）由 CPU 寄存器返回，最后根据返回地址值，返回到 A()函数的“②”处继续执行 A()函数的代码，即把返回值 65 赋值给变量 x。接着回到 A()函数，继续执行 A()的剩余代码。A()的栈帧处于栈顶位置，如图 3-14 和图 3-15(d)。

（5）当 A()执行结束，A()函数栈帧内容相继退栈，A()栈帧释放，根据返回地址返回到主函数 main()的“①”处，继续执行 main()函数后面代码。main()的栈帧处于栈顶位置，如图 3-14 和图 3-15(e)。

（6）当 mian()函数执行完毕，这个程序执行结束，main()栈帧释放，然后整个调用栈销毁。

图 3-15 是用一个简图来说明上面例子**调用栈**的建立和销毁过程，每个栈帧简单用一个方块表示。

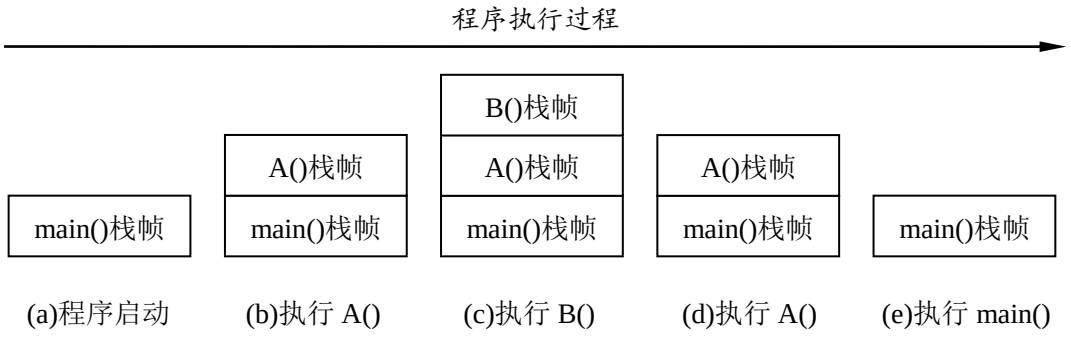
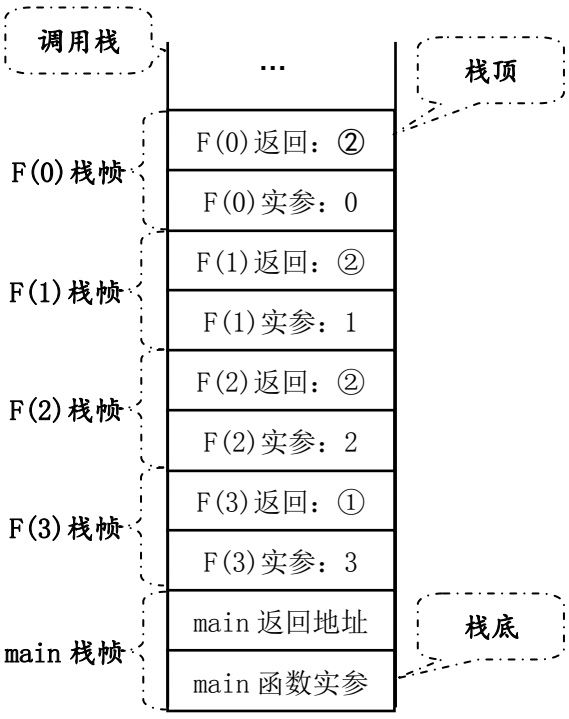


图 3-15 嵌套调用调用栈变化示意图

3.递归函数调用的实现原理

搞清了一般函数调用的实现原理，再学习递归调用的实现原理就很简单了。**递归调用与一般函数调用的内部实现原理相同**。不同的只是递归调用时，每次调用的都是函数自身，即每次递归调用执行的代码都相同（调用自身），但每次调用时的函数参数都不同，函数的本地变量值也可能不同。下面以求 3 的阶乘为实例，说明递归调用的内部实现原理，程序代码如下。主函数中调用 F(3)，求 3 的阶乘，F(3)执行结束，返回到 main()的“①”处继续执行。F(3)执行到“②”处，递归调用 F(2)，F(2)执行结束，将返回到“②”处，计算 3*F(2)。同样 F(2)执行到“②”处，递归调用 F(1)，F(1)执行结束，返回到“②”处，计算 2*F(1)。F(1)执行到“②”处，递归调用 F(0)，F(0)执行结束，返回到“②”处，计算 1*F(0)。图 3-16 和图 3-17 说明程序执行时调用栈和栈帧的建立和释放过程，同时也演示了递归的正向递推调用和方向回归合成解的过程。为了简化在图中省略了相关寄存器内容；另：这两个函数都没有本地变量，也省略不画。



【主函数代码】

图 3-16 递归调用调用栈示意图

```
int main()
{
      F(3);
①:    cout<<“OK”;
      return 0;
}
```

【求阶乘函数代码】

```
int F( int n )
{
     if(n==0)
        return 1;
     else
②:      return n*F(n-1);
}
```

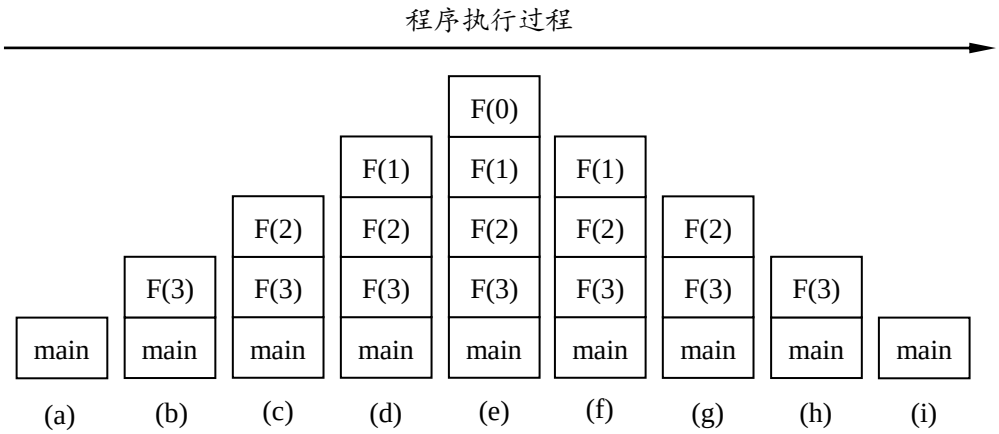


图 3-17 递归调用调用栈变化示意图

下面解释递归调用时调用栈和栈帧的建立和释放过程：

（1）启动程序，建立函数调用栈，main()函数栈帧入栈。执行 main()函数代码。如图 3-16 和图 3-17(a)。

（2）main()函数中调用求阶乘函数 F(3)，求 3 的阶乘。F(3)的实参 3，返回地址“①”等入栈形成 F(3)的栈帧。然后转入 F(3)执行，F(3)栈帧处于栈顶，为活动记录。如图 3-16 和图 3-17(b)。

（3）F(3)执行到“②”处，需要计算 3*F(2)，因而递归调用求阶乘函数 F(2)，求 2 的阶乘，执行的代码与 F(3)相同，只是实参为 2。此时，F(2)的实参 2，返回地址“②”等入栈形成 F(2)的栈帧。然后转入 F(2)执行，F(2)为当前正在执行的函数，栈帧处于栈顶位置。

如图 3-16 和图 3-17(c)。

（4）同理，F(2)执行到“②”处，要计算 2*F(1)，递归调用 F(1)，建立 F(1)栈帧，转入 F(1)执行。如图 3-16 和图 3-17(d)。

（5）F(1)执行到“②”处，要计算 1*F(0)，递归调用 F(0)，建立 F(0)栈帧，转入 F(0)执行，图 3-16 和图 3-17(e)都表示函数 F(0)正在执行时调用栈的情况，此时 F(0)栈帧为活动记录，处于栈顶位置。

（6）F(0)是递归的出口，当 F(0)执行结束时，正向递推过程结束，接下来开始反向回归合成解的过程。取出返回值 1，返回地址“②”。释放 F(0)的栈帧，返回到其调用者 F(1)的“②”处继续执行 F(1)余下部分代码，即利用 F(0)的返回值 1，计算 F(1)的返回值 1*F(0)=1*1=1。此时 F(1)为正在执行的函数，其栈帧为活动栈帧，处于栈顶位置。如图 3-17(f)。

（7）当 F(1)执行结束，取出 F(1)的返回值 1，返回地址“②”。释放 F(1)的栈帧，返回到其调用者 F(2)的“②”处，继续执行 F(2)余下部分代码，即利用 F(1)的返回值 1，计算 F(2)的返回值 2*F(1)=2*1=2。此时 F(2)为正在执行的函数，如图 3-17(g)。

（8）当 F(2)执行结束，取出 F(2)的返回值 2，返回地址“②”。释放 F(2)的栈帧，返回到其调用者 F(3)的“②”处，继续执行 F(3)余下部分代码，即利用 F(2)的返回值 2，计算 F(3)的返回值 3*F(2)=3*2=6。此时 F(3)为正在执行的函数，如图 3-17(h)。

（9）当 F(3)执行结束，取出 F(3)的返回值 6，返回地址“①”。释放 F(3)的栈帧，返回到其调用者 main()的“①”处，继续执行 main()余下部分代码，此时 main()为正在执行的函数，如图 3-17(i)。

（10）main()函数执行结束，其栈帧退栈释放，整个程序执行结束，这个程序的函数调用栈也随之释放。

递归调用的返回地址在很多情况下是相同的，如上例中，F(2)、F(1)和 F(0)的返回地址都是“②”，但递归开始函数的返回地址一定是不同的，如上例中 F(3)是返回到 main()函数中的“①”。

我们可用使用一些调试工具来跟踪和查看程序执行时调用栈的实际工作状态。比如 VC6.0 在调试模式下就可以查看调用栈的情况，图 3-18 就是 VC6.0 跟踪求阶乘递归函数，调用 Fact(6)时调用栈情况的截图。图中显示当前正在执行 Fact(2)函数，Fact(2) 的栈帧处于栈顶位置，为活动栈帧。通过设置断点，然后单步执行，还可以看到每次函数调用，栈帧的建立和释放过程；递归的正向递推过程和反向回归过程。

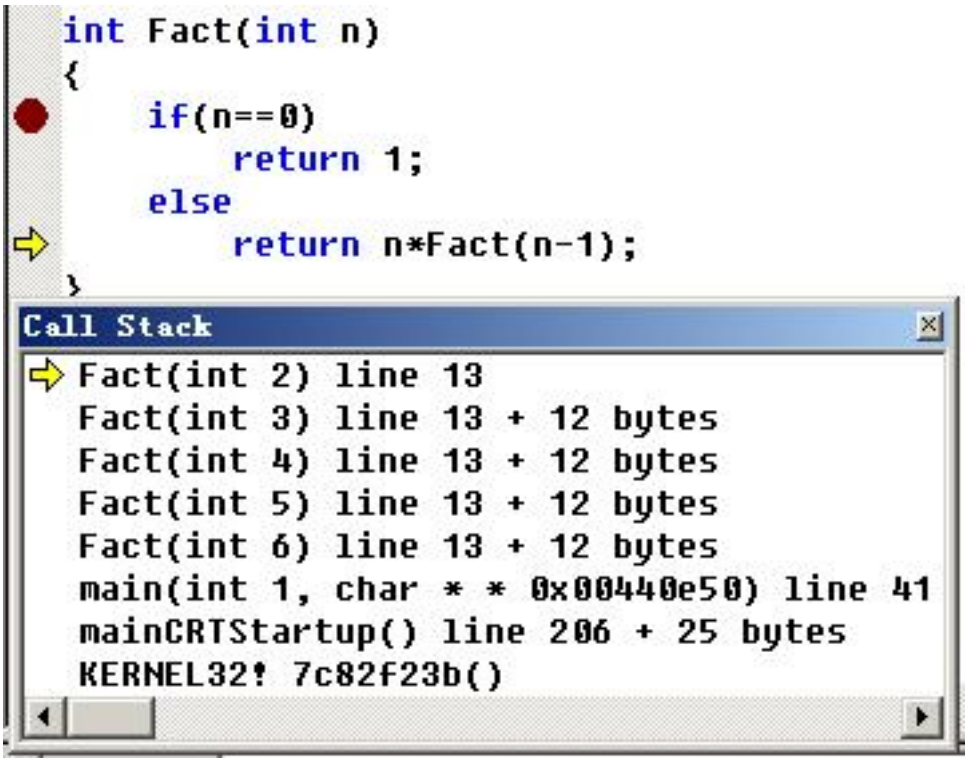


图 3-18 VC6.0 中递归调用调用栈情况截图

许多材料上会笼统的说递归调用会耗费内存空间，什么原因呢？从上面的讨论和例子中可以清楚看出这一点。系统为每个线程开辟的函数调用栈空间大小是有限的。每一次函数调用都要在调用栈上建立一个函数调用栈帧。栈帧的大小由函数参数的个数及大小、返回地址、相关寄存器值和本地变量的个数及大小共同确定。其中，返回地址和相关寄存器值占用空间相对固定。但函数参数和本地变量的个数和大小在不同的函数中都是不同的。不管怎样每建立一个栈帧就会消耗掉调用栈上的部分存储空间。所以，递归调用中，正向递推的过程是不能无限加深的，递归每加深一次，至少会新建一个栈帧，消耗掉调用栈的部分空间，这样迟早会造成函数调用栈的溢出（调用栈满）。特别是栈帧较大时，更容易溢出。许多读者在平时运行一些软件时，所见到的“stack overflow”或“栈溢出错”就是因为递归调用或嵌套函数调用层次太深，造成函数调用栈满溢出的情况。

3.3.3 递归程序的阅读和理解

在实际应用中，常常要求能够快速地模拟递归程序的执行过程，并给出其运行结果，因此，需要有更快捷、使用的方法。下面介绍一种依据递归内部实现原理，图形化的递归程序阅读方法。

这种方法采用图形化的方式描述程序的运行轨迹，从中可较直观地描述各调用层次及其执行情况，是一种比较有效的递归程序阅读和理解方法。事实证明，理解这一方法对后续有关内容的学习大有帮助，若能结合后面介绍的树、二叉树的遍历过程来讨论，收获会更大。

这个描述方法中以每次调用的函数为主要结点，因为每次递归调用的函数名相同，我们用实参加以区分。然后按“先主后次”的次序给出程序的运行轨迹。这里的“主”指两条线，一是正向递推线，一是反向回归线，结点之间用有向边（弧）进行连接，表明执行次序，这是核心部分，它们分别描述了调用点、返回点和执行路径。这里的“次”是指，在正向递推线上可能还进行了其它的计算、打印结果等操作，如果有就要用相应的图形符号加以表示；在反向回归线上，可能有函数返回值或修改了函数参数，也要以相应图形符号标记。最后对草图加以整理即可得到直观的运行图。详细步骤如下：

（1）根据调用情况，画出每个调用函数作为图的主要结点，以实参进行区分，用有向边连接各个结点，表明调用层次和调用点；

（2）从递归出口点开始，逐次回归返回，返回线路也用有向边链接，直到递归起始函数，给出整条反向回归线。在绘制返回线时要特别注意函数的返回点；

（3）在正向递推线相关位置，画出函数执行的其它计算、打印等结果；在反向回归线的相关位置，画出函数的返回值或修改的返回参数；

（4）重绘上面得到的草图，使更加美观和直观。从这个图上就可以清晰的见到递归的运行轨迹和程序的执行结果。

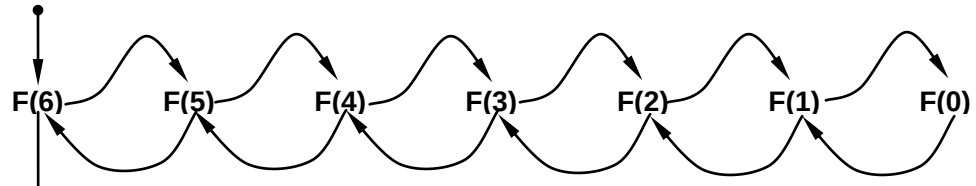
下面分线性递归和树形递归两类情况给出递归路线图的实例。

1. 线性递归执行路线图

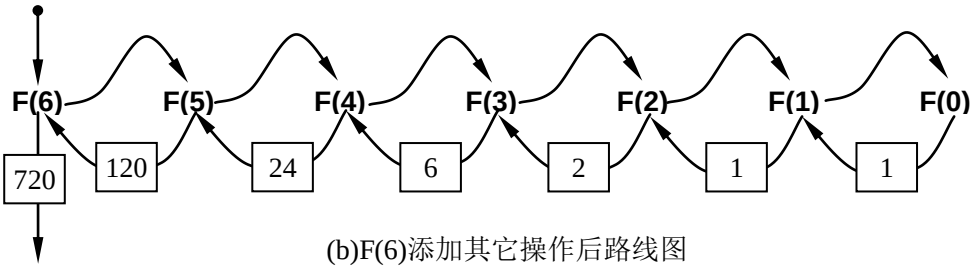
递归函数体中，如果只有一次递归调用，这样的递归叫做**线性递归**。线性递归执行路线图的正向递推线是一条“线”，画起来相对简单。

【例 3.6】画出求阶乘函数 F(6)的执行路线图，并给出执行结果。

【分析】阶乘函数体内只有一次递归调用，属于线性递归。我们先画出正向递推线和反向回归线两条主线，如图 3-19(a)所示。假定我们在正向调用路线上没有其它的计算和处理；返回线路上需要添加函数返回值，整理后得到完整的执行线路图 3-19(b)。



(a)F(6)执行主线



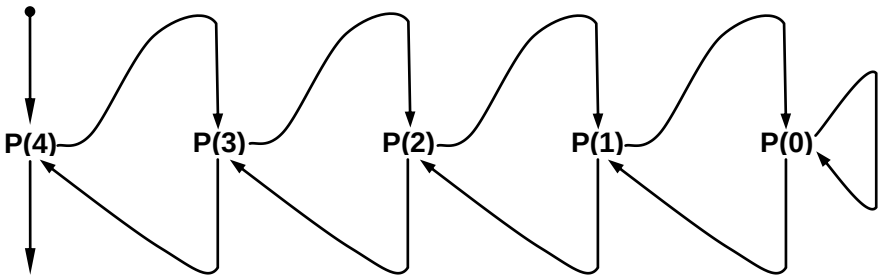
(b)F(6)添加其它操作后路线图

图 3-19 阶乘 F(6)执行路线图

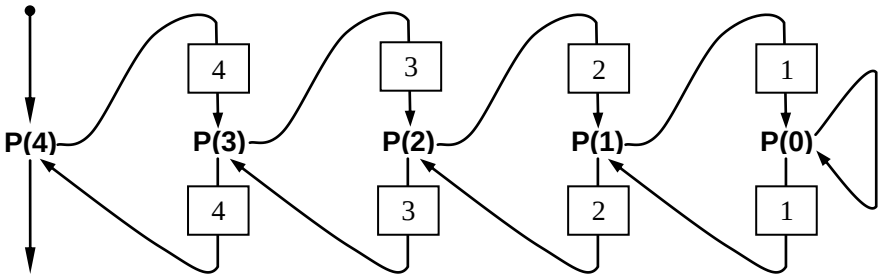
【例 3.7】对下面的程序代码，画出 P(4)的执行路线图，并给出执行结果。

```
void P(int n)
{
    if(n>0)
    {
        printf("%d",n);
        P(n-1);
        printf("%d",n);
     }
}
```

【分析】这也是一个线性递归函数，先画出执行主线如图 3-20(a)所示。再在执行路线上添加上两个打印操作，整理后的最终执行路线图，如图 3-20(b)。由路线图可见执行结果：4 3 2 1 1 2 3 4。



(a)P(4)执行主线



(b)P(4)添加其它操作后路线图

图 3-20 P(4)执行路线图

2. 树形递归执行路线图

函数体内有两次或两次以上的递归调用，我们称这样的递归为**树形递归**。我们在画这种递归的执行路线图时，如果只保留正向递推线路，省略其它所有东西，将得到一棵递归调用树，这也是其名称的由来。在画完整的执行路线图之前，我们先画出其递归调用树，有助于我们理解递归执行过程，然后再在树上添加反向回归路线，以及其它计算和处理，这样比直接画出执行路线图要简单的多。

【例 3.8】对下面的程序代码，画出调用 AB()的执行路线图，并给出执行结果。

```
void A( )
{     cout<<'A';
}
void B( )
{
     cout<<"B";
     A( );
     cout<<"B";
}
void AB( )
{
     cout<<"AB";
     A( );
     B( );
     cout<<"AB";
}
```

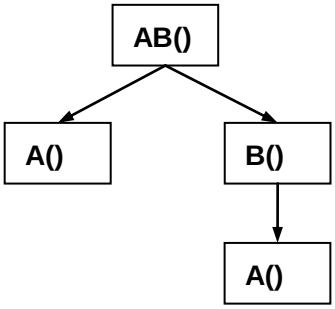
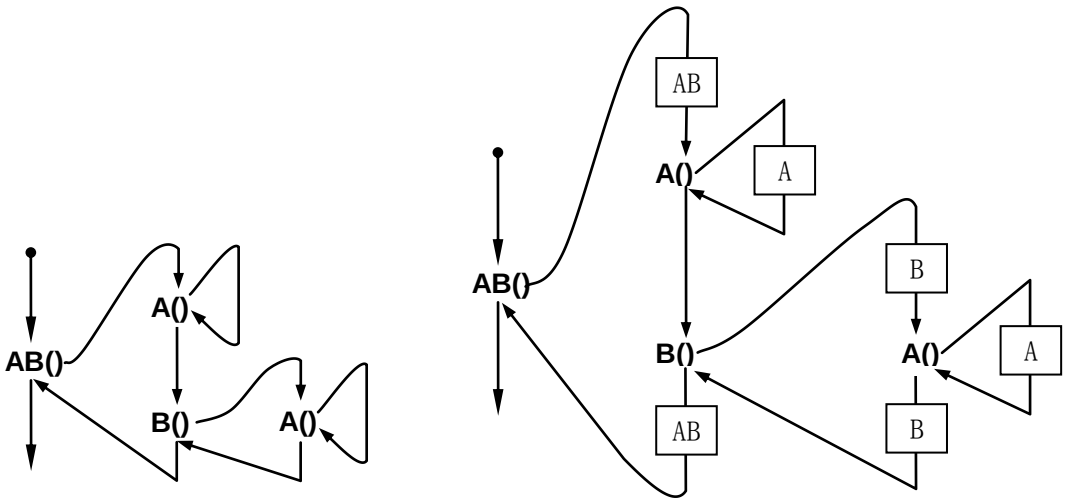


图 3-21 函数 AB( )的调用树

【分析】这不是一个递归函数，但函数 AB()中有两次函数调用，也可以画出函数调用树，如图 3-21 所示。执行主线如图 3-22(a)，添加其它处理后的执行路线图如图 2-22(b)。



(a)AB()执行主线　　(b)AB()添加其它操作后路线图

图 3-22 AB( )执行路线图

【例 3.9】对下面的程序代码，画出调用 P(4)的执行路线图，并给出执行结果。

```
void P( int w )
{
     if( w>0 )
```

```
    {
        P( w-1 );
        cout<<w;
        P(w-2);
    }
}
```

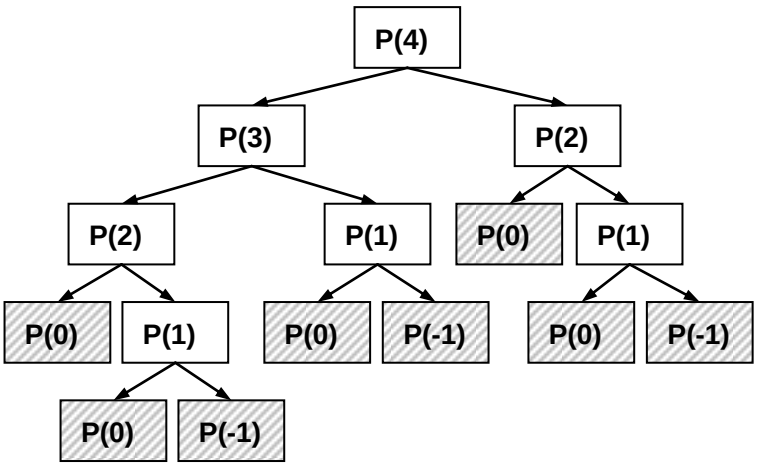


图 3-23 函数 P(4)的调用树

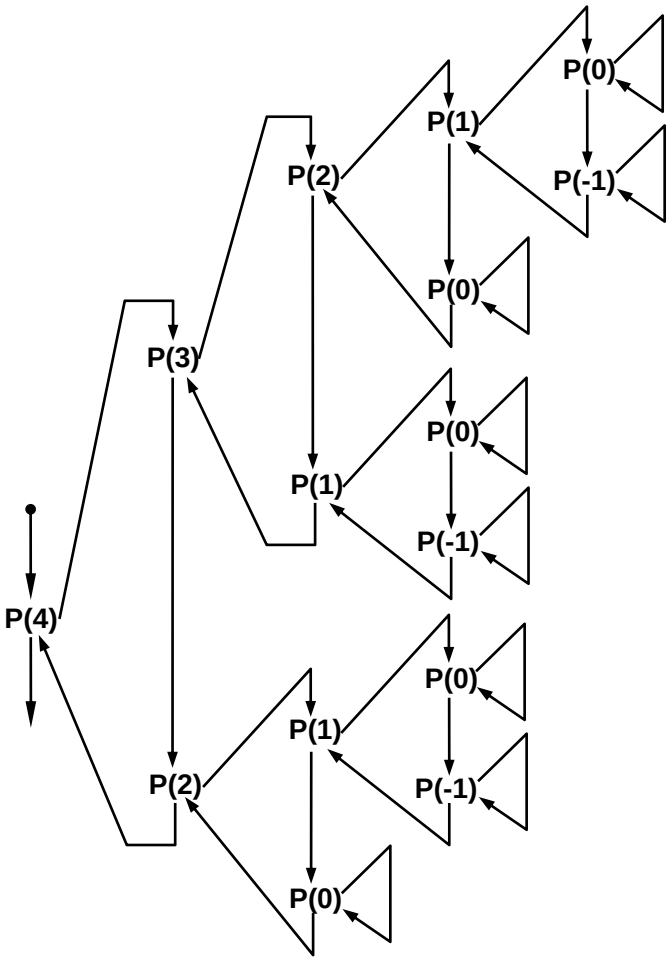


图 3-24 P(4)执行主线

【分析】函数体中有两次递归调用，所以这是一个树形递归，递归调用树如图 3-23 所示，其中阴影部分结点为递归基本问题，或递归出口。

事实上，我们根据源代码和调用树即可画出递归调用主线，如图 3-24 所示。在执行主线图上添加上其它计算和处理得到最终的执行路线图，如图 3-25 所示。

执行结果为：1 2 3 1 4 1 2。

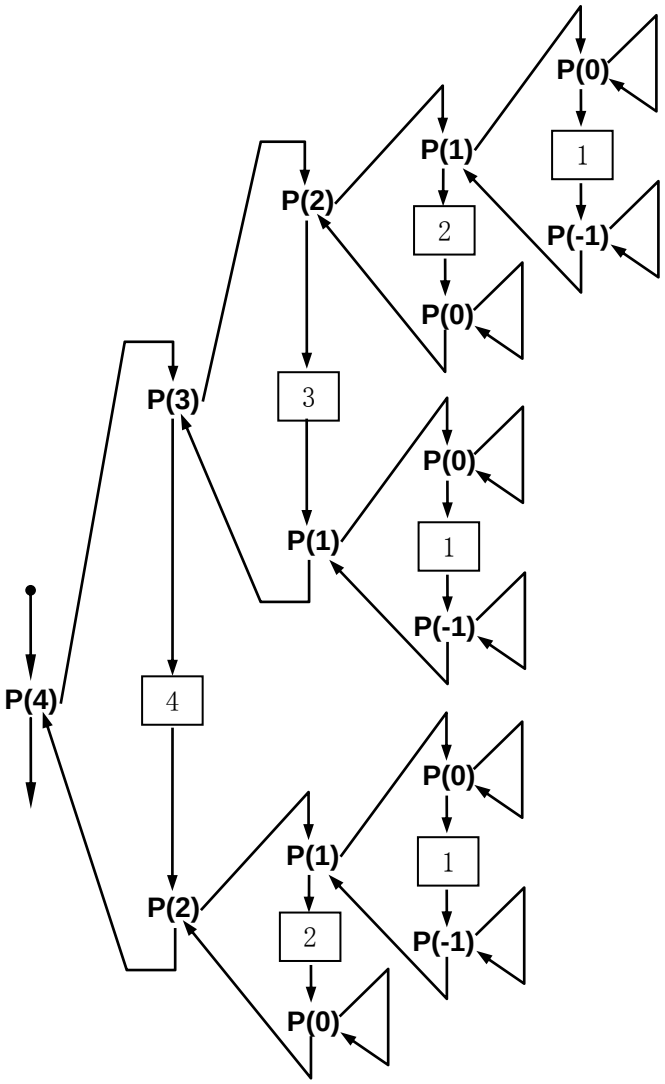


图 3-25 P(4)执行路线图

【例 3.9】对下面的程序代码，画出调用 F(5)的执行路线图，并给出执行结果。

```
int F( int n )
{
     if( n<=2 )
         return 1;
     return 2*F(n-1) + 3*F(n-2);
}
```

【分析】本题有点类似 Fibonacci 函数，在函数体中有两次递归调用，属于树形递归，且在返回线路上要做计算处理。递归出口条件为 n<=2，所以基本问题为 F(1)和 F(2)，返回值都为 1。画出的递归调用树如图 3-26。执行主线如图 3-27。执行路线图如图 3-28。

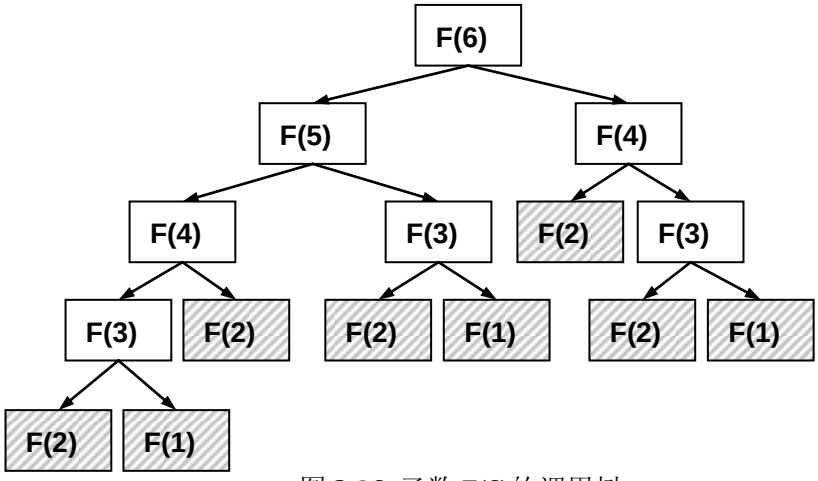


图 3-26 函数 F(6)的调用树

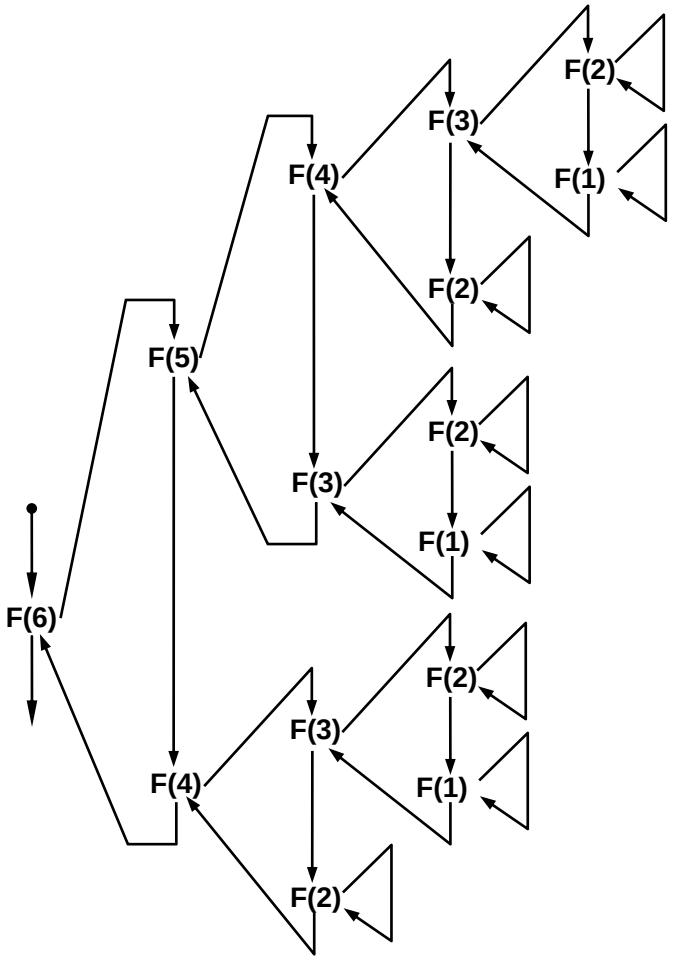


图 3-27 F(6)执行主线

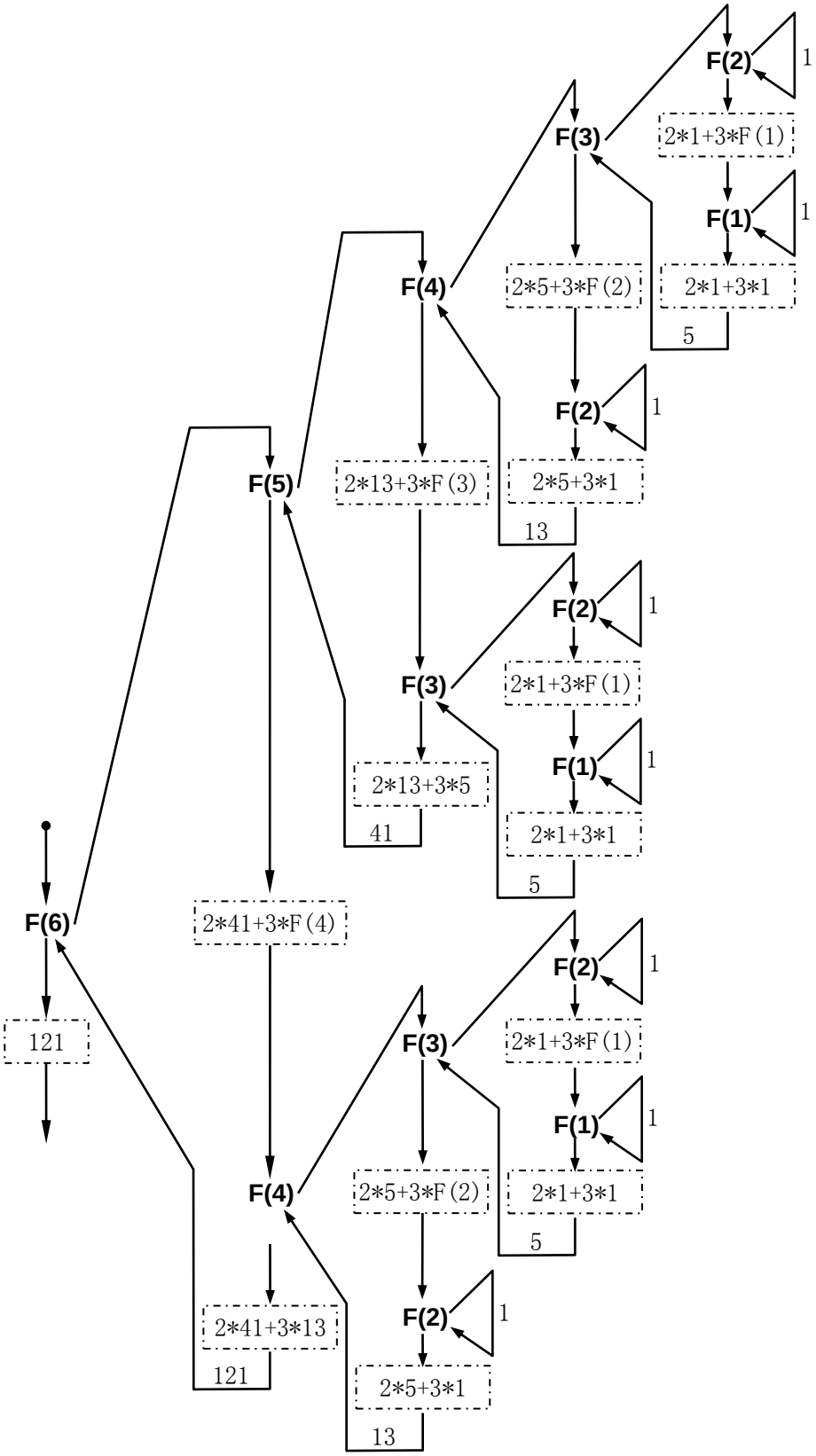


图 3-28 F(6)执行路线图

从例 10 中 F(6)的执行过程可以看出树形递归的执行效率较低。F(1)有三次重复调用，在函数调用栈中经过三次建立栈帧和释放栈帧的过程；F(2)有五次重复调用，经历五次栈帧建立和释放过程；F(3)三次，F(4)两次。每次建立和释放栈帧都是有时间消耗的，所以程序执行效率较低。例 9 也有同样的问题。

可见，只要是树形递归，就会存在同样函数的多次重复调用（参数和本地变量都相同），造成重复建立和释放同样的栈帧，使程序执行效率变低。同样，如果递归层次较深，参数或本地变量较大，容易造成栈溢出。

3.3.4 递归程序编写

不是所有的问题都可以用递归方法求解的，我们在 3.3.1 小节讨论了递归求解的条件，只有满足这三个条件的问题，才可以编写递归程序求解。

这样，当我们面对一个问题并试图用递归求解时，首先就要对照递归的三个条件进行分析，判定递归求解是否可行。一旦确定用递归求解，我们需要做好下面的工作：

（1）分析出正向递推分解子问题的规律，并给出形式化的表示。比如求阶乘的递推公式为：n!=n*(n-1)!，求 Fibonacci 数的递推公式为:Fib(n)=Fib(n-1)+Fib(n-2)。对于较复杂的问题这一步可能是相对困难的工作。

（2）分析出递归出口条件和基本问题的解。有的递归只有一个出口，有的递归会有多个出口。比如，求阶乘的出口条件为 n=0，只有此一个出口，对应的基本问题为 Fact(0)=0；求 Fibonacci 数,就有两个出口,分别为 n=0 和 n=1,对应两个基本问题 Fib(0)=0 和 Fib(1)=1。

（3）分析反向回归合成解的规律。有的递归只需简单地返回值；但有些递归不仅要返回值，而且要用返回值进行其它计算和处理；简单的递归甚至不要求返回值。例如求阶乘需要根据 Fact(n-1)的返回值做 n*Fact(n-1)的计算;求 Fibonacci 数要根据 Fib(n-1)和 F(n-2)的返回值，计算 Fib(n-1)+Fib(n-2)。

前面我们已经举了很多简单递归程序的例子，下面我们再举两个稍微复杂的一点的例子，对照上面的方法进行分析来编写求解程序。

【例 3.10】一只小猴第一天摘了若干桃子，当即吃掉一半，还没过瘾，又多吃了一个；第二天又将前一天剩下的桃子吃掉一半,又多吃一个;以后每天都吃掉前一天剩下的一半另加一个。到第十天早上猴子想再吃时发现只剩下一个桃子了。问第一天猴子共摘了多少个桃子？

【分析】

假定求解函数名为：int MonkeyPeach(int t)，函数返回值为桃子数，t 为天数。

（1）分析递推公式。这个问题我们可以从第十天的桃子数，倒退求出第九天的桃子数，一直到第一天。假定今天的桃子数为 n_2，前一天的桃子数为 n_1，而今天的桃子数为昨天吃剩下的，即 $n_2=n_1-(n_1/2+1)$，得到 $n_1=2*(n_2+1)$。这样第十天为 1，第九天为 4，第八天为 10，第一天为 1534。递推公式写成函数形式即为：return 2*(MonkeyPeach (t+1)+1);

（2）第十天为递归出口条件，对应的基本问题为 $n_2=1$。代码为：if(t==10) return 1;

（3）反向合成原问题解。对这个问题很简单，只要调用 MonkeyPeach (1)，即求出第一天的桃子数。

也有人把第一天看作 10，第十天看作 1，这样上面的公式、出口条件、原问题解的表示

都要做相应变更。

【算法描述】

```
int MonkeyPeach(int t)    //主调函数中调用 MonkeyPeach(1)
{
    if(t==10)
        return 1;  //递归出口
    else
        return 2*(MonkeyPeach(t+1)+1);  //递归调用
}
```

【例 3.10】河内塔问题（Problem of Hanoi Tower）。一个底座上有 a、b 和 c 三根柱子，a 柱子上放置有 n 片直径不同的圆盘，大盘放底下，小盘放上面。现在要求把所有圆盘全部从 a 移动到 c 柱子上。游戏规则：一次只能移动一片圆盘；移动过程中大盘不能压在小盘上面；移动过程可以借助第三根柱子。

【分析】假设我们用 1 到 n 对盘片进行编号，数字越大表示圆盘越大，1 号盘最小，n 号盘最大。我们以 4 片盘的移动为例分析问题的解法，如图 3-29 所示。

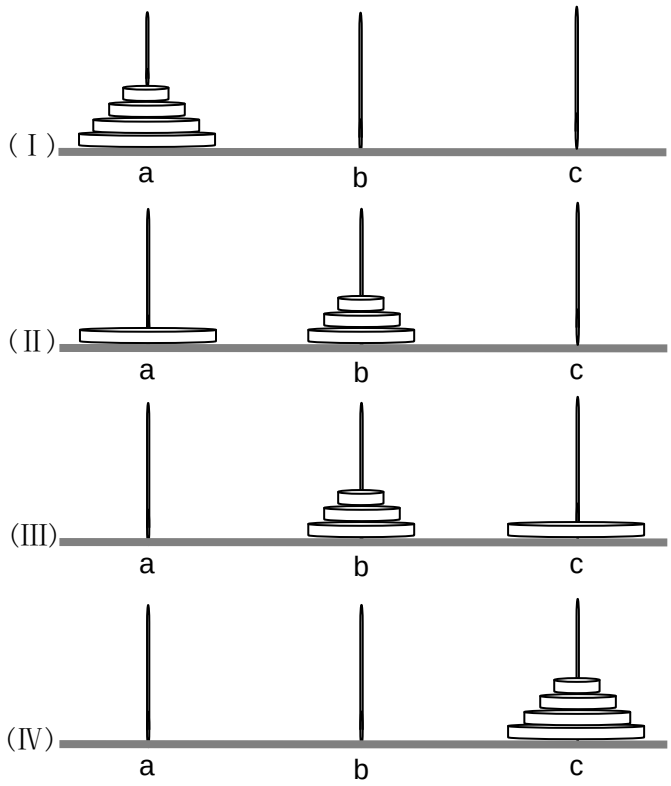


图 3-29 河内塔示意图

递归求解分析如下：

（1）若想把 4 片盘全部从 a 移到 c，关键是要把 4 号盘先移到 c 柱；若要移动 4 号盘，必须把压在其上的 1、2、3 号盘移走，且最好放到 b 柱子上，如图 3-29(Ⅱ)。

（2）此时，a 柱上只有 4 号盘，c 柱为空，4 号盘可以从 a 移到 c，如图 3-29(III)。

（3）最后，再设法将 b 柱上的 3 片盘移动到 c 柱，这样就实现了全部的移动，如图 3-29(IV)。

通过上面的分析可见，移动 4 片盘的问题可以分解为 3 个子问题来解决，即 2 个移动 3 片盘的子问题，和 1 个移动 1 片盘的子问题。这里 1 片盘的移动是直接可求解的基本问题，不需再分解。同样，3 片盘的移动问题可按同样的方法分解为 3 个子问题解决：2 个移动 2 片盘问题，和 1 个移动 1 片盘问题。显然，可以用递归分解，每次分解后问题求解难度降阶。

对于 n 片盘的移动，也是这样，现把 a 柱上 n-1 片盘移到 b 上；n 号盘从 a 移到 c 上；再把其它 n-1 片盘从 b 移动 c。n-1 片盘采用同样方法分解，如此下去，直到 1 片盘移动的基本问题。

假设函数为 void Hanoi(char a, char b, char c, int n)，其中 a、b、c 为柱子名，n 为盘片数，含义为把 n 片盘从 a 柱移动到 c 柱，中间借用 b 柱；另定义一函数表示 1 片盘的移动，void Move(a,n,c)，表示将编号为 n 的 1 片盘子从 a 柱移到 c 柱。根据上面的分析得到：

（1）正向递推规律。n 片盘移动分解 3 个子问题。用上面定义的函数表示就是：Hanoi(a,c,b,n-1)，Move(a,n,c)，Hanoi(b,a,c,n-1)。即现把较小的 n-1 片盘，经 c 柱，从 a 柱移动到 c 柱；n 号盘从 a 柱移到 c 柱；b 上的 n-1 片盘，经 a 柱，从 b 柱移到 c 柱。

（2）递归出口条件。n=1，即 1 片盘子的移动，对应的基本问题 Move(a,n,c)。

（3）反向回归求解原问题。在主函数中调用 Hanoi(a,b,c,n)即可，n 用实际的盘片数。

【算法描述】

```
void Hanoi(char a, char b, char c, int n)
{
    if(n>=1)
    {
        Hanoi(a,c,b,n-1);  //将 a 上的 n-1 片判移动到 b，c 为借用盘
        Move(a,n,c);       //表示移动一个圆盘，从 a 到 c。
        Hanoi(b,a,c,n-1);  //将 b 上的 n-1 片判移动到 c，a 为借用盘
    }
}
```

Move()函数可以很简单，比如只是一条简单的打印语句：

```
void Move(char a,int n, char c)
{
    cout<<"移动圆盘 "<<n<<"："<<a<<"-->"<<c<<endl;
          //表示移动一个圆盘 n，从 a 到 c。
}
```

这个问题的来源有不同的传说，有说来自印度寺庙的游戏，有说来自欧洲的修道院，有说来自越南河内。传说中这个底座是镶满各种宝石的，柱子和圆盘都是金子的，a 柱上有 64 片盘。如果一个人把所有盘从 a 柱移到 c 柱，一说是完成后这些财宝就归它了；一说是完成之时就是宇宙毁灭之时。事实上我们可以计算出移动的总次数，n 片盘需要移动的总次数为 2^n-1 次。64 片盘的移动次数是 18,446,744,073,709,551,616 次。如果一个人每秒钟移动一

片盘，大约需要 584,942,417,355.07 年，可见一个人有生之年是无法完成的。即使每秒中移动 10 亿片盘的计算机，也需要大约 584.94 年才能完成。

前面我们分析了递归调用的不少缺点，那么我们为什么还要使用它呢。从例 9 和例 10 我们就能大概看出一些端倪，这两个看上去有点复杂的问题，用递归编程求解只不过 3 行代码就解决了，这正式递归的魅力所在。接下来我们总结一下递归调用的缺点和优点：

【递归调用的缺点】

空间效率不高。每次递归调用都要在函数调用栈上建立一个调用栈帧，耗费一些内存空间，递归深度越深，栈帧就越多，占用的内存越多。容易导致“栈溢出”错误；

时间效率不高。树形递归调用，即在递归函数体内有多次递归调用时，会出现多次重复调用同样的函数（相同代码、相同参数、相同本地变量），而每次调用函数都有建立栈帧和释放栈帧的过程，花费一定的时间，重复调用造成相同的栈帧被重复的建立和释放，完成同一任务花费重复的时间。

【递归调用的优点】

递归调用有完善的数学理论支撑，其正确性可以用数学归纳法加以证明。

递归容易编程实现，我们只需按照数学定义就可以写出正确的递归程序，有时甚至还未弄清楚问题的求解机理。

递归编程求解问题，代码简短，便于阅读，容易理解。甚至可称之为“优美”。往往几行代码就可以解决很复杂的问题，前面我们已经看到这样的例子，在后面学习树、二叉树和图的遍历算法时还可以充分见证这一点。

递归代码健壮性和可维护行好。

递归通过让计算机做更多的事情，而让人做更少的事情。

3.3.5 递归程序转换和模拟

由前述内容可知，递归方法的优点和缺点都非常明显，我们在解决实际问题时怎样来扬长避短呢？通常的做法是一开始用递归设计算法，并用递归方法编程实现，然后再寻求把递归程序转换为**等价的**非递归程序。当然，转换的前提是确保两者的等价性，且程序性能得到提升。还有，除了递归的上述缺点外，有些程序设计语言不支持递归，比如 Fortran 语言，就必须把递归算法转换为非递归实现。

下面我们分两类情况讨论递归程序转换为非递归程序的方法。

1.直接用循环（迭代）方法转换

有些类型的递归程序可以直接用循环（迭代）方法进行等价转换，这种转换会使得程序的时间和空间性能都得到极大提高，因为减少了递归调用中栈帧的建立和释放过程，节省了时间和空间。

（1）尾递归

函数体中只有一次递归调用，且调用发生在函数的最后，即函数最后一条语句是递归调用，递归调用返回时，不需进行其它运算，这种递归叫**尾递归**（tail recursion）。包括有些借助辅助函数和辅助参数可以转换为尾递归的函数。例如求阶乘函数，本身不是尾递归函数，因为递归调用返回后，还要计算 n*Fact(n-1)，但借助一个辅助参数可以转化为尾递归调用，代码如下：

【尾递归调用求阶乘代码】

```
int FactTail(int n, int result)
{
    if(n==0)
        return result;
    else
        return FactTail(n-1, n*result);
}
```

用辅助参数保存累乘的结果，开始调用时 result=1，即 FacTail(n,1)。转换为循环方法也是要借助一个辅助变量来保存累乘的结果，代码如下：

【阶乘的循环方法代码】

```
int Fact1(int n)
{
    int result=1;  //保存累乘结果
    int i;
    for(i=n;i>=1;i--)
        result=i*result;
    return result;
}
```

（2）单向递归

单向递归指函数体中有两次以上递归调用，但指递归的过程总是朝着一个方向进行，即递归函数的参数只由主调函数确定，递归调用不影响函数参数，称这种递归叫做单向递归。比如求 Fibonacci 数函数就属于单向递归。上面讨论的为递归是单向递归的一个特例。单向递归借助辅助参数和辅助函数也可以转换为尾递归调用，只是可能要借助多个辅助参数。比如求 Fibonacci 数的递归函数转换为尾递归调用，代码如下：

【尾递归调用求 Fibonacci 数代码】

```
int FibTail(int n, int k, int f1, int f2)
{
    if(n==k)
        return f1;
    else
        return FibTail(n,k+1,f1+f2,f1);  //尾递归
}
```

我们引入了 3 个参数，k 初始化为 1，每次递归 k 加 1，以实现从 Fib(2)往 Fib(n)方向累加；f1 相当与 Fib(n-1)，初始化为 1；f2 相当于 Fib(n-2)，初始化为 0。这个尾递归函数初始调用形式为 FibTail(n,1,1,0)。同样我们引入相关参数，可将其转换为循环实现，代码如下：

【循环方法求 Fibonacci 数代码】

```
int Fib1(int n)
{
    int f1=1,f2=0,tem,i;
```

```
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
    {
        for(i=2;i<=n;i++)
        {
            tem=f1;
            f1=f1+f2;
            f2=tem;
        }
        return f1;
    }
}
```

2.递归的栈模拟

递归的栈模拟就是我们自己定义一个软件栈来模拟递归调用时系统使用函数调用栈建立栈帧和释放栈帧的过程。理论上讲这种方法可以将任何递归函数转换为非递归函数。但是必须明确的是通过这种转换后，程序代码会变得“面目全非”，代码的可读性和可维护性都将很大程度上变差，且容易出错。由于还是使用栈，还是有调用时的入栈操作（类似系统调用的建立栈帧）和返回时的出栈操作（类似系统调用时的释放栈帧），因而转换后程序的时空性能并没得到改善，甚至比系统调用时的性能更差。唯一的好处可能就是可以开辟更大的软件栈空间，突破系统对函数调用栈的空间限制。所以，使用这种方法转换时，一定要分析经过优化后，程序的时空性能的确有改善，否则不建议做这种转换。它的真正作用在于深入研究递归的实现机理。下面仅给出这种转换的步骤：

（1）定义一个软件栈 S，初始化为空栈；

（2）在调用函数的入口处设置一个标号地址（不妨设为 L_0）。系统调用时这一步是系统通过指令指针自动完成的，转为人工调用时，必须有一个地址，明确函数从哪开始执行。

（3）模拟递归的正向递归过程。即模拟系统调用时，建立栈帧的过程，可用以下等价操作替换：

①保留现场：在栈 S 的栈顶，将函数实参，返回地址，函数本地变量值入栈。

②转入函数入口点执行，即执行 goto L_0;

③在函数原来的递归返回处设置标号地址 L_i（i=1,2,3,…），即原递归调用的返回地址。有些递归只有一个返回点，有些递归有几个返回点，根据实际情况进行设置。

④根据需要，如果函数有返回值，需要人工定义一个回传变量，模拟函数返回时取出返回值，传送到相应位置。这个过程在系统调用中也是自动完成的。

（4）模拟反向回归过程。即模拟系统调用时，释放栈帧的过程，可以用以下等价操作替换：

如果栈 S 不空，则依次执行下列操作，否则结束函数执行，返回。

①回传数据：函数值返回的值保存到回传变量中；函数参数回传的从相应参数化取

出。系统调用中此步自动完成。

②回复现场：从栈顶取出返回地址和需要回传的参数。本次调用的相关信息退栈。这个操作相当于系统调用时释放一个栈帧。

③返回，按取出的返回地址，执行 goto X，假定取出的返回地址放在 X 中。

（5）对其它非递归函数的调用和返回可以照搬上述步骤。

按照上述 5 个步骤我们就可以把任何递归程序机械地转换为非递归程序。但上述方法在实现时，还有一些具体问题，比如函数参数、返回值、本地变量可能有不同的数据类型，再加上返回地址类型。如果只定义一个软件栈，那么就要先定义一个包含所有需要数据类型的结构体，将上面的各个数据变为结构的一个分量来处理；还有一种办法是定义几个栈，一个栈处理一种类型的数据。不管那种方法都比系统调用来的复杂，都需要人工去处理和完成。前面已经分析过，这种方法并没太多实用价值，这里不再举具体的例子，我们后面在学习树和图的遍历算法时，会给出一些通过软件栈将递归遍历函数转换为非递归函数的例子。

本章小结

栈和队列都是运算受限的线性表，是软件设计中最常用和最基本的数据结构。

栈是只能在一端（顶端）进行插入和删除的线性表，因而具有先进后出特性。对栈的存储与线性表类似，有顺序存储和链式存储两种，由此而得到顺序栈和链栈两种存储结构。

顺序栈和顺序表结构相同，并将表的后面部分设置为栈顶，其运算的时间复杂度都较好。然而，由于许多情况下事先难以估计所用栈的最大规模，因而需要采用链栈。

采用链表形式存储栈所得到的链栈，其栈顶设在表头部分，实现运算较为方便。

队列是只能在一端（尾端）进行插入，另一端（队头）进行删除的线性表，因而具有先进先出特性。对队列的存储也有顺序存储和链式存储两种，由此而得到顺序队列和链队列两种存储结构。

与顺序表结构中仅设置一个指针所不同的是，顺序队列中设置了头尾两个指针，分别指示队列的第一个元素之前的位置和最后一个元素的位置。由于在此结构中执行入队和出队时，其头尾指针均是往后移，故可能会出现“假溢出”的问题。为此，通过将数组的首尾元素看作是相邻元素而引入循环队列结构。然而，由于循环队列中满和空的状态的判断条件相同，故需要采用相应的处理方法，其中的一个解决方案就是在队列为“满”时还能有一个元素空间为空闲状态。

链队列是采用链表存储的队列，其中将队头元素设置在链表的表头，将队尾放在链表表尾，并采用带头结点的结构形式，以便不同位置的运算。